Deutsches Zentrum
für Luft- und Raumfahrt
German Aerospace Center

Frankfurt University Of Applied Sciences

Faculty: Computer Science and Engineering

High Integrity Systems

Master's Thesis

Response Time Analysis of Tasking Framework Task Chains

Author:	Jathin Sreenivas
Matriculation Number:	1321762
Examiner:	Prof. Dr. Christian Baun
Co-examiner:	Prof. Dr. Thomas Gabel
Supervisor:	Dr.-Ing. Zain Hammadeh
Submission Date:	August 25th 2022

I herewith formally declare that I, Jathin Sreenivas, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I marked and separately listed all of the literature and the other sources I employed when producing this master's thesis, either literally or in content. The drawings or images in this master's thesis were created by myself or provided with a corresponding source reference. This thesis has not been handed to any other examination authority in the same or similar form.

Frankfurt Am Main, August 25th 2022 Jathin Sreenivas

Abstract

Multi-core processors have been increasingly utilized in general computing and modern embedded applications for their potential to maximize system throughput. Parallel frameworks allow programmers to make the most of parallelism without having the burden of understanding the underlying architecture. However, real-time systems comprise tasks governed by stringent timing requirements, which the parallel frameworks do not support. There is a need to analyze a computation model that adapts both advantages. The analysis of parallel real-time applications modeled as Directed Acyclic Graph (DAG) tasks scheduled on multi-core platforms has been intensively studied in recent years. A real-time task can be modeled as periodic and sporadic tasks. In recent years, sporadic tasks have been modeled as periodic by considering the maximum arrival frequency as the period. Current studies provide an analysis of the challenges faced for scheduling real-time tasks modeled as DAG tasks on multi-core processors where all the subtasks (fragments of the task) are consigned to and executed by the worker threads of a thread pool by restricting the maximum parallelism at any point of execution by the number of threads in the thread pool.

However, the existing work dispatches the subtasks to the threads in a non-deterministic way, i.e., the execution order of the subtasks is not contemplated. The work done here proves that the intra-task priorities have a notable impact on the worst-case response time. Furthermore, it confirms that the upper bound of response time computed by modeling sporadic tasks as periodic is pessimistic. An algorithm is introduced that allows analyzing a safe upper bound for the response time by controlling the execution order. Moreover, a function is utilized to model sporadic tasks without maximal arrival frequency to achieve a less pessimistic result.

An analysis is made to derive a worst-case response time for a task set scheduled by a preemptive global fixed-priority scheduler, wherein each task has intra-task priorities assigned. The work is further extended by providing experiments with randomly created DAG tasks showing that the proposed method outperforms the current state-of-the-art methods.

Contents

1. Introduction

Parallel processing of tasks using current multi- and many-core architectures has proved advantageous in accelerating task execution, system scalability, and low power consumption. The significant ramp-up in multi-core processors has led to dramatic changes in programming paradigms. The usage of the sequential paradigm has been reduced over the past years due to the advantages of parallel paradigms. The parallel frameworks, e.g., OpenMP, are up-and-coming for exploiting multi-core processor systems' advantages. These frameworks have the capability to split a given task into a set of smaller parallel execution fragments and dispatch them to workers of a thread pool.

Nonetheless, parallel scheduling of tasks on a multi-core processor is not designed to satisfy time-sensitive requirements. The migration from the sequential model on single-core platforms to a parallel model on multi-core platforms poses many challenges to real-time systems design. Therefore, an analysis of how to ensure maximum attainable parallelism is required. In real-time systems, tasks are governed by stringent timing requirements, which introduces a challenge in scheduling as many as possible sequential real-time tasks on multi-core processors. Therefore, using the advantages of parallel paradigms in multi-core processors to operate real-time systems fulfilling the stringent timing requirements is a promising study to improve the scheduling of real-time applications.

Parallelism frameworks, for instance, OpenMP, provide parallel design pattern interfaces which provide programmers to parallelize applications using the interfaces, wherein a parallel task is broken down into small fragments and is then executed on a run time environment that consists of a scheduler that dispatches these fragments onto a thread pool consisting of worker threads for execution. This procedure was adapted to real-time systems as well [53] [51].

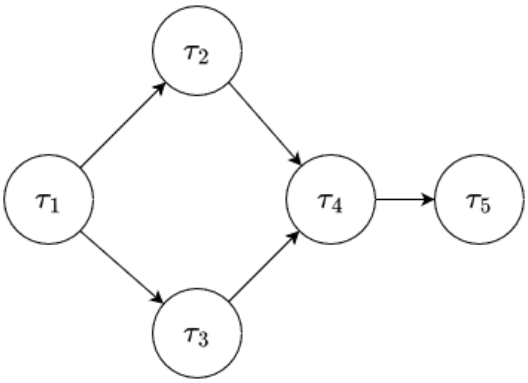


Figure 1.1.: Example DAG created using Parallel Frameworks API

A code snippet that utilizes the parallelism APIs provided by a parallel framework is shown at Listing 1.1. *spawn* creates a node or a subtask to be executed on a thread pool. *sync* allows creating dependencies between tasks, i.e., waits for the execution of subtasks that are passed as arguments. The resulting outcome is a Directed Acyclic Graph as shown in Figure 1.1.

Listing 1.1: Example usage of parallel framework's API

```
void TaskA() {
    // Create Source
    subtaskA1();
    // Create children of source
    Subtask A2 = spawn(subtaskA1);
    Subtask A3 = spawn(subtaskA3);
    // Wait for subtasks A2 and A3
    sync(A2, A3);
    Subtask A4 = spawn(subtaskA4);
    // Wait for A4
    sync(A4);
    // Create Sink
    subtaskA5();

}
```

Real-time application tasks can be categorized into periodic and sporadic tasks. Two consecutive activations of the same task are separated by a specific time known as the period. However, in the current state-of-the-art methods, the sporadic tasks are integrated by treating them as periodic tasks by considering the minimum distance between two activations as the period itself, i.e., maximum arrival frequency [51].

Parallel real-time applications are possible to be modelled using Directed Acyclic Graph, the state of the art methods show a safe upper bound for the response time of a DAG task [24] [51] [5] [38]. When scheduling a parallel DAG task, at some time point, many nodes of the task are eligible for execution and are to be executed on a multi-core system. Existing state of the art methods [20] [39] [45] [29] [6] [51] do not consider the order in which the execution of the nodes must take place.

1.1. Use case scenario

Considering a real-world real-time application example, the Figure 1.2 shows an optical navigation system for a spacecraft [22], which is a part of the Autonomous Terrain-based Optical Navigation (ATON) project at the German Aerospace Center (DLR). The analysis of the optical sensor data and a trigger on the system's estimated position are done in real-time. Cameras are periodically triggered, and once all the input camera data is received, the image analyzer is executed, i.e., a combination of a periodic and a sporadic task. As shown in Figure 1.2 two cameras are triggered periodically using a timer, and then the camera data

is sent to the analyzer components. The feature tracking component estimates a relative movement in the camera data, whereas the crater navigation matches the craters on the moon in the camera data images. Both components' output is then sent to the navigation filter, which estimates an output position, then sent to the logger and the flight controller.

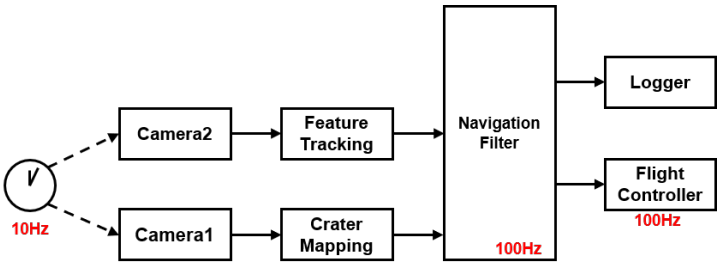


Figure 1.2.: Optical navigation sub-system

The optical navigation system can be modeled as a Directed Acyclic Graph as shown in Figure 1.3 wherein the source of the DAG is the timer task which is a periodic task, camTask1 and camTask2 are dependent on the timer task, i.e., only after execution of timerTask, camTask1 and camTask2 can start their execution. Similarly, featureTask and CraterTask can start their execution only after the completion of camTask2 and camTask1, respectively. navTask must wait for the execution of both featureTask and craterTask for its execution and the same execution model is followed for the rest of the tasks. The last task, i.e., sinkTask, is added as a dummy node to ensure the DAG single source-sink model is maintained (detailed explanation in chapter 4). The dummy node is just added to maintain the model, and it does not contain any instructions to be executed. It takes zero time to complete.

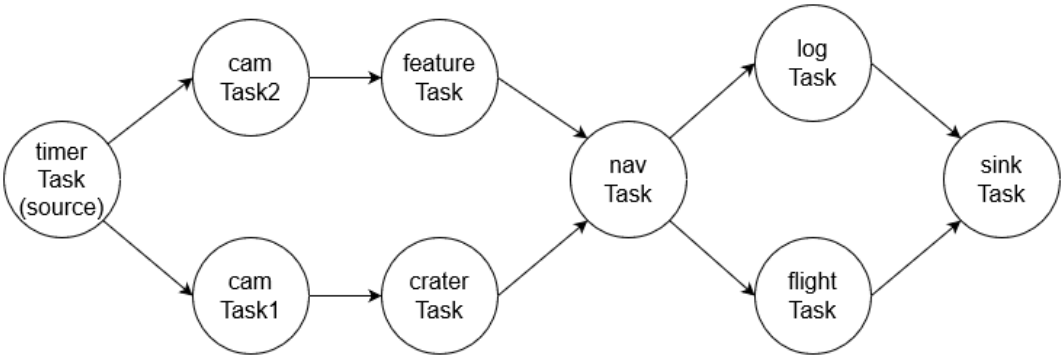


Figure 1.3.: Optical navigation sub-system modeled as a DAG

1.2. Contribution and Structure

The thesis aims to prove that the execution order of eligible nodes on a multi-core system significantly impacts the upper bound of the response time. An algorithm is introduced to compute response time when the order is assigned, i.e., nodes with priorities. Moreover, methods to assign a specific execution order by assigning intra-task priorities are illustrated. Finally, a function is described to model sporadic tasks using the actual arrival frequencies. The contributions made in the thesis can be summarized as follows:

- Firstly, the chapter 2 provides a real-world use case as to where the findings of the thesis can be applied to.
- The work done in the thesis is an extension of a work that already exists, which is explained in the chapter 3. Furthermore, it provides an analysis of other existing research that solves a similar problem.
- The chapter 4 is where the contribution begins. This chapter introduces a real-time task model comprising multiple subtasks, which are then dispatched and executed among the worked threads of the thread pool in a non-preemptive way. This chapter also explains how a preemptive global fixed-priority scheduler schedules a task set on a multi-core platform.
- After the explanation of the task model, the model is used across the thesis and the chapter 5 explains the motivation for the usage of sporadic tasks in real-time applications and also provides a function to handle the arrival frequencies of the sporadic DAG tasks in a less pessimistic approach.
- The chapter 6 begins with providing motivation that an order of execution, results in a shorter response time upper bound. Moreover, this chapter is completed by deriving a method to compute a new response time bound for a single DAG task.
- The chapter 7 illustrates algorithms to assign intra-task priorities to the DAG task.
- The chapter 8 explains the implementation of all the designs explained above. In addition, random Directed Acyclic Graph generators are described, which are used in the chapter 9 to experiment with multiple random DAG designs and configurations to prove the findings of the thesis.
- The chapter 9 shows experiments that investigate the usage of assigning priorities by comparing the response time bound for randomly generated Directed Acyclic Graph tasks and then proves that the new response time bound for a single Directed Acyclic Graph task is valid and can be replaced with the current state of the art response times.
- Finally, chapter 10 and chapter 11 are used for summarizing all the chapter's findings and mentioning the following possible research that can be studied to extend the work explained, respectively.

2. Tasking Framework

There exists a great demand for computing resources in modern space applications for increasing computational requirements of onboard data processing and complex control algorithms. Firstly, multi-core platforms provide high performance with low power consumption compared to the uni processors with high frequencies and therefore are promising to fulfill the computational requirements. Nevertheless, it is not straightforward to implement applications that execute in parallel. Furthermore, sensors are slow and cannot be on par with the computing resources. Self-suspending processes read from sensors make timing more complicated and present high pessimism and, thus, high over-provisioning [22] [44].

"Tasking Framework is a multi-threaded execution platform and a software development framework that is developed using abstract classes with virtual methods, which facilitates the implementation of space applications as event-driven task graphs" [22]. Tasking Framework splits the computations into small tasks scheduled on the input data's availability. Real-time applications are often described as a graph illustrating the software components and their dependencies. The implementation of Bi-spectral Infrared Detection (BIRD) and Attitude and Orbit Control System (AOCS) is as shown in Figure 2.2. During the BIRD mission, the control modules were computed in a fixed order and at fixed intervals. The total computation time was the sum of the waiting time for sensor data and the latency required to ensure the entire data arrival before computation begins. This model caused timing problems due to latency overestimation [1], leading to corrupted data and faulty behavior to the Attitude and Orbit Control System (AOCS) system. The BIRD mission hence provides a need for a conceptual framework that allows sharing of resources based on certain configurations for all flight phases [1].

2.1. Task-Channel Model

The Tasking Framework uses the task-channel paradigm [17], to separate the data and the functionality, which improves the re-usability of code. In the task-channel model, A task is a stateless executable program, along with its memory and I/O ports. In contrast, a channel is a message queue that connects the output port of a task to an input port of another task. In Tasking Framework, a channel represents a data container managed through the task object and can be represented by an interface that serves as an interface between tasks and between software inputs and outputs, as illustrated in the Figure 2.1 [22].

The Tasking Framework was created to handle data-flow-oriented applications, wherein the system's input data must be identified and processed to produce the required outputs. Using this approach, the application is modeled as a series of sequential operations executing in a

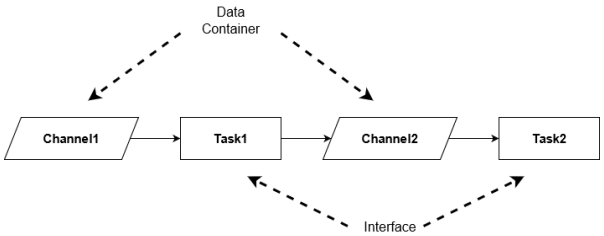


Figure 2.1.: Task Channel Model

particular order. The Tasking Framework employs this design pattern to create an interface that is structural and not reliant on data availability but rather on the flow of execution [22]. The framework can be seen as an abstraction layer similar to the operating systems that control the whole process in an abstract, generic, and deterministic way. All APIs provide a high level of abstraction, making them independent of data's availability and the processed task.

2.2. Execution model

Tasking Framework consists of the execution platform and the application programming interface (API). Using Tasking Framework, applications are implemented as a graph of tasks, connected via channels, and each task has one or more inputs. Periodic tasks are connected to a source of events to trigger the task periodically. In Tasking Framework, an instance of a task τ is activated only when all its inputs are activated [22]. For instance, in the Figure 2.2, $F(A)$ is activated only after receiving the input 1 from the channel $MsgA$, which in turn receives an input from the $SensorA$, the channel $MsgA$ acts as an interface between the $SensorA$ and $F(A)$. A way to execute a task immediately can be done by making one of its inputs final. If a final task is activated, the task will run without considering the other inputs. For instance, in the Figure 2.2, $TaskE$ is immediately activated when the Timer activates the final input 0 regardless of the other inputs.

The Tasking Framework's sequence diagram is illustrated in Figure 2.3. When a message from a sensor is received, the main execution thread uses the channel class method $push()$ to notify the dependent inputs. In the scenario where all task inputs are activated, the Tasking Framework will instantly inform a thread to run the task instance by invoking $perform()$. The framework's scheduler adds the task to the queue right away. The job will begin as soon as the required resources, such as a CPU and memory, are available; otherwise, the task will be queued [22] [44].

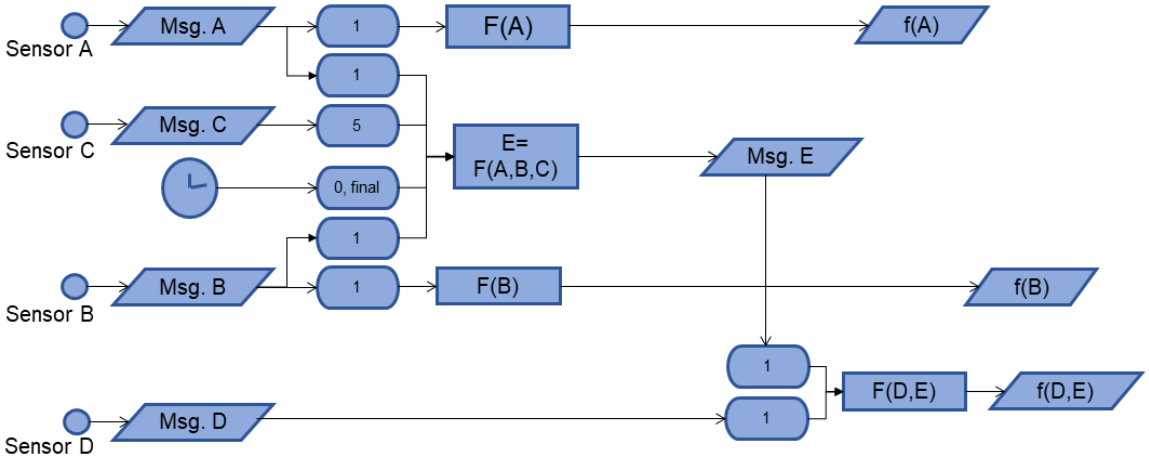


Figure 2.2.: BIRD - Attitude and Orbit Control System (AOCS) and Tasking Framework Elements

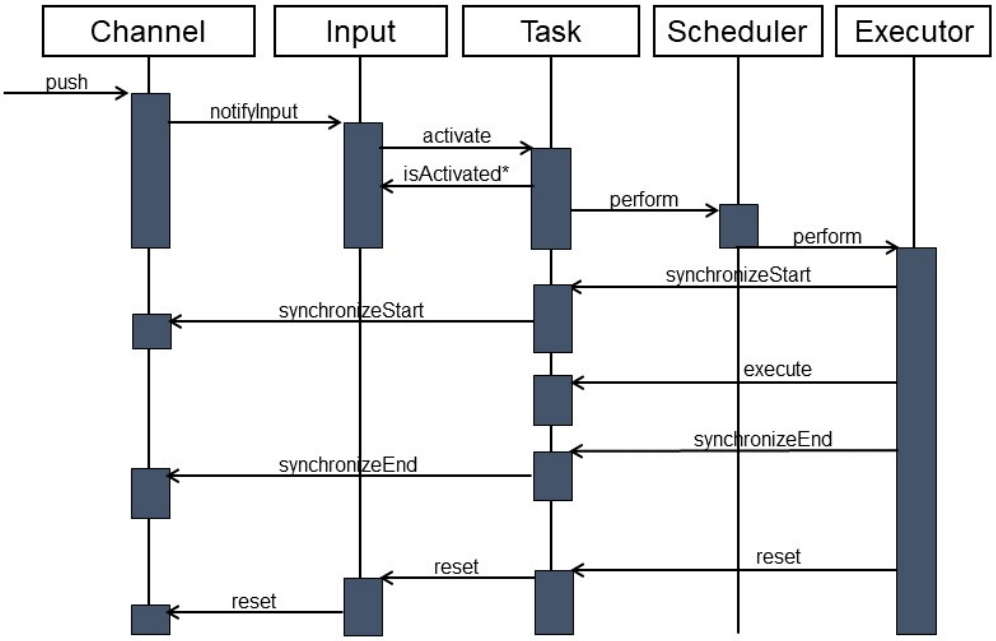


Figure 2.3.: Tasking Framework Sequence Diagram

2.3. Application Model

A data-flow application, when implemented in Tasking Framework, is modeled as a directed graph, where data is processed by tasks and then forwarded to subsequent tasks in a pipelined manner. The instructions of a task do not have to wait for the preceding tasks to complete its execution but can be executed once the data are available, i.e., event-driven. The Tasking Framework API provides abstract classes to design the applications as a directed graph of tasks and channels. In other words, the API does not depend on any run-time data. The Figure 2.4 represents the Figure 2.2 as a directed graph using tasks and channels [22].

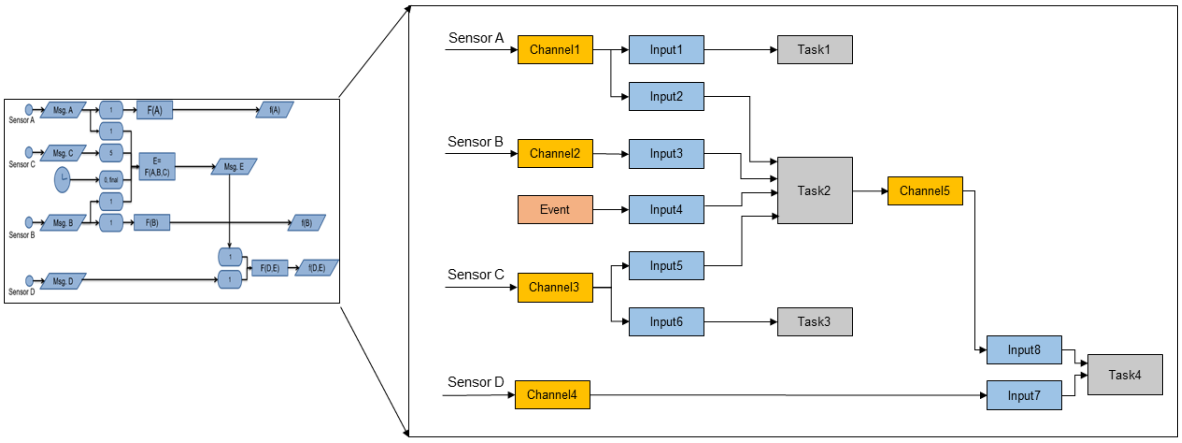


Figure 2.4.: BIRD - AOCS as realized in Tasking Framework

2.4. Applications

The section highlights some applications wherein the tasking framework is used at German Aerospace Center (DLR). Firstly, in Autonomous Terrain-based Optical Navigation (ATON) [54], the tasking framework was used for the implementation of the functionalities that were used for modeling the components using tasks and channels to connect tasks, wherein the channels are activated periodically using events. ATON was developed using four threads to execute the software on a prototype flight computer. The Scalable On-Board Computing for Space Avionics (ScOSA) [42] which is an application that tests the onboard computer architecture that is based on re-configurable interconnected commercial off-the-shelf processors [1], wherein tasking framework was used as middleware by providing an API to develop applications to execute on Scalable On-Board Computing for Space Avionics (ScOSA). An example of such an application that is executed on Scalable On-Board Computing for Space Avionics (ScOSA) using tasking framework is the Onboard Data Analysis, Real-time Information System (ODARIS) [32] and Rendezvous Navigation [48] [1].

2.5. Timing analysis of Tasking Framework task chains

In order to prove the real-time capability of Tasking Framework, timing guarantees have to be computed on the tasks using Tasking Framework. Abaza et al. proposed in [2] a worst-case execution time analysis to compute the WCET of subtasks in Tasking Framework task chains. This work aims to use the WCET of subtasks as input to compute the response time of tasks.

The DAG as shown in Figure 2.5 that is modeled as a DAG task model based on the tasking framework task chains as shown in Figure 2.4. Where τ_0 represents a dummy source with zero execution time, it is added to maintain the model of the DAG. τ_1, τ_2, τ_3 and τ_4 represents the subtasks Task1, Task2, Task3 and Task4 respectively. τ_4 is dependent on both τ_0 and τ_2. Finally, an extra sink is added with zero execution time to maintain the model of the DAG.

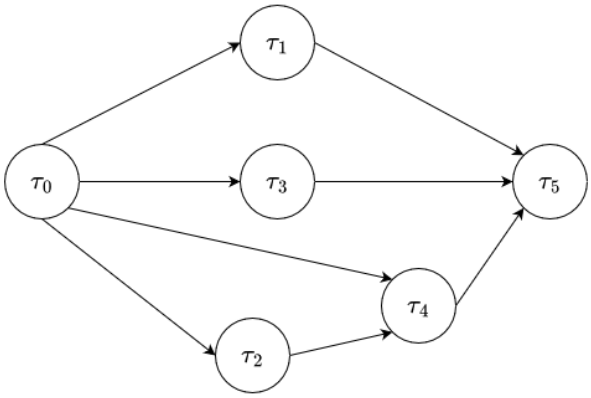


Figure 2.5.: Tasking Framework application DAG

3. State Of the Art

The parallelism frameworks have been extensively studied in the past years [33] [46]. However, the literature on the usage of parallelism frameworks for real-time systems is only a few [51]. The following task models stand out in this literature for parallel real-time scheduling. Firstly, the fork-join task model, wherein the tasks are modeled as a sequence of alternating sequential and parallel executions. For parallel execution, all the jobs have the same worst-case execution time [35]. The execution of the fork-join task model is the work done at [50] which generalizes the usage of the fork-join task model by allowing a segment to have an arbitrary number of parallel jobs with different execution times, but there is no response time analysis Global Fixed Priority scheduling [50]. The work done at [43] provides a feasibility analysis for Global Fixed Priority scheduling and considers the maximum interference the critical path (The path with the longest execution time) suffers either by jobs that are not in the critical path or other higher priority tasks. Next, for the DAG models, the tasks are modeled as Directed Acyclic Graphs, where each node in the graph represents a sequential computation, and the edges represent a dependency between the jobs. Response time analysis for DAG models for Global Fixed Priority are mentioned at [39] [45] [29] [16]. However, this literature does not specify the execution order of the nodes in the DAG. The OpenMP tasking model [52], which models a real-time system as a single OpenMP application with multiple DAG tasks. However, the nodes of a task cannot be preempted during their execution when using a limited preemptive scheduler, but higher priority tasks can be executed between two nodes of a graph but do not use thread pools for execution. Gang scheduling [31] proposes using worker threads of a thread pool. However, all the worker threads execute for the same amount of time.

3.1. Directed Acyclic Graph task model

Apart from all the models mentioned above, the model presented in this thesis is an extension of the model presented at [51], wherein, Directed Acyclic Graph task model consisting of fine-grained task computation or subtasks or nodes is scheduled among the worker threads of a thread pool and scheduling of these fine-grained parallel real-time tasks on thread pools have limited parallelism by the number of threads in its thread pool. The model uses two schedulers, one a real-time operating system scheduler which allows for preemptive scheduling of threads of each task's thread pool on the cores. Moreover, a runtime scheduler allows integration into each task to dispatch the subtasks among the worker threads [51]. It also provides the analysis for the safe upper bound of the response of tasks where the threads are scheduled using a preemptive global fixed-priority scheduler while a work-conserving scheduler dispatches the nodes or subtasks. However, in this model, for a single parallel DAG

task running on a multi-core platform, there exists a case at some point during execution, the number of eligible nodes of the task is more than one, the work done at [51] does not specify the order in which the execution of these eligible nodes must take place, i.e., assumes a non-deterministic execution order. However, the interference for a single task is considered a safe upper bound, but the bound is pessimistic.

3.2. Intra-task priorities

The classic response time bound used in literature [51] [45] [29] [6] is calculated using scheduling algorithms which executes a set of eligible node in a non-deterministic manner. The work done at [20] does specify the order of execution, but this information is not considered in the computation of a response time. The literature [25] and [24] do consider intra-task priorities and also provide an analysis of the computation of a safe upper bound response time; however, the execution model that is used is different from the model that is used in the thesis. The execution model of the literature [25] [24] assumes that each node of a Directed Acyclic Graph (DAG) has its very own thread in a thread pool for its execution, and the execution of the subtasks can be preempted not only by the higher priority tasks but also by, the higher priority subtasks of the same task, i.e., uses a preemptive scheduler for scheduling both tasks and subtasks.

3.3. Priorities Assignment

The literature presented at [34] [30] [25] [24] provides priority assignment strategies that can be used for the DAG task models such that the response time bound is reduced as much as possible. However, this is not within the scope of the thesis. The most commonly used methods are illustrated in the chapter 7.

4. Preliminary

The chapter introduces a Directed Acyclic Graph (DAG) task model that dispatches and executes its fine-grained computations using the worker threads of its thread pool. The executions are done using two different schedulers one a real-time operating system scheduler that schedules threads of each task's thread pool preemptively on the cores of the multi-core system and second uses a run-time system scheduler for each thread pool that dispatches the subtasks among the work threads of the thread pool. Each task thread pool thread is scheduled using a preemptive global fixed priority, whereas the subtasks are dispatched by a non-preemptive work conserving scheduler.

4.1. System Model

This section defines a system model for a real-time application executing on a multi-core system. When executed on a multi-core system, a real-time application is executed as a single process, consisting of N parallel real-time tasks. A parallel real-time task is modeled as a Directed Acyclic Graph's (DAGs). A process comprises of a task set $\tau = \{\tau_1, \tau_2.....\tau_N\}$ with N parallel real-time sporadic DAG tasks assumed to be indexed in priority order with τ_1 having the highest priority, which is scheduled on a multiprocessor system composed of m identical processors with uniform memory access, assuming all time units are positive multiples of the system clock. Each DAG task τ_i. When a task τ_i is executed, it releases a set of sporadic jobs or subtasks that are dispatched among and executed by the thread pool workers in a work-conversing environment. The number of threads in a thread pool is limited by m, restricting the number of parallel execution at any point by the number of worker threads in the thread pool. Moreover, at any point in time, the maximum parallelism of the DAG tasks is limited by the number of threads in its thread pool and not by the number of processors [51].

A real-time Directed Acyclic Graph (DAG) task τ_i is represented as a 4-tuple $\tau_i = (G_i, \Phi_i, D_i, T_i)$ where T_i is the minimum time interval between two activations of the task (Period), D_i represents a timing window by which the task must complete its execution (Deadline). Φ_i represents the thread pool the task will use to execute its subtasks, and G_i represents the task modeled as a DAG.

Definition 4.1.1. *A Directed Acyclic Graph (DAG) $G_i = (V_i, E_i)$, where $V_i = \{v_{i,1}, v_{i,2}....v_{i,n}\}$ is the set of vertices and $E_i \subseteq V_i \times V_i$ is the set of directed edges of the DAG [51].*

Definition 4.1.2 (Worst Case Execution Time (WCET))**.** *Worst-case execution time is the maximum length of time a task takes to execute on a specific hardware platform [41].*

Each vertex $v_{i,j} \in V_i$ is a subtask of the task τ_i that comprises of a Job which is a sequential computation that takes $c_{i,j}$ time to execute in worst case, characterised as Worst Case Execution Time (WCET). Throughout the paper, the subscript i of the vertices is omitted for ease and simplicity. A subtask, vertex, or node represents a job comprising of WCET and a priority assigned to it. A priority of a subtask v_i is denoted by $p(v_j) = \{p_i | 0 <= p_i < n\}$, where a subtask with the highest priority is assigned the value 0, formally v_i has a higher priority than v_j if $p(v_i) < p(v_j)$. An edge $E_{i,j} = (v_i, v_j)$ represents a precedence relation between v_i and v_j, where the subtask v_j is dependent on the subtask v_i i.e, v_j can start its execution only when v_i completes its execution, formally v_i is the predecessor or ancestor of v_j or v_j is the successor or descendant of v_i. For a subtask $v_{i,j}$ the set of ancestors are represented by $ancestors(v_i) = \{v_j \in V_i | (v_j, v_i) \in E_i\}$ and the set of descendants by $descendants(v_i) = \{v_j \in V_i | (v_i, v_j) \in E_i\}$.

The Directed Acyclic Graph (DAG) presented in this model are assumed to operate using the AND semantic, i.e., A subtask is ready to execute only when all its ancestors have completed their execution. Furthermore, when a subtask completes its execution, all its descendant subtasks are ready. A vertex that does not have any incoming edges, i.e., an empty ancestor set, is called the source (v_{source}) of the DAG and a vertex that does not have any outgoing edges is called the sink (v_{sink}), each DAG is assumed to have only one source and sink. For real-time tasks with multiple sources or sinks, an additional single dummy source or sink with zero WCET is added along with its corresponding edges to complete the DAG.

The Figure 4.1 DAG shows a real-time task τ_i modeled as a G_i with 7 subtasks, and 8 edges, with v_1 being the source of the DAG and v_7 the sink. The numbers inside the parentheses indicate the subtasks' WCETs. The subtask v_1 has a WCET of 2 and has the highest priority. The edge from $v_3 \rightarrow v_5$, represents a dependency that v_5 can start its execution only when v_3 completes its execution. The ancestors of v_7 are $ancestors(v_7) = \{v_4, v_5, v_6\}$, thus v_7 can start its execution only when all its ancestors have completed their execution. The subtasks v_2 and v_3 have the same ancestor v_1, and v_2 has a higher priority than that of v_3.

4.2. Execution Model

For a given DAG task τ_i, the subtasks of the task are dispatched among and executed by $\mu_i = |\Phi_i|$ worker threads. The worker threads in the thread pool Φ_i share the same priority as that of the task τ_i and are preemptively scheduled on the processors by the operating system scheduler. And these threads are scheduled on a preemptive global fixed-priority scheduler while a non-preemptive work-conserving scheduler dispatches the subtasks, i.e., a subtask of a task can be preempted only by a task of higher priority and cannot be preempted by a higher priority subtask of the same task. The number of processors constrains the maximum number of threads in each pool, i.e., $\forall i : \mu_i <= m$, ensuring that there is a limit for the number of parallel execution by the number of threads μ_i in the thread pool Φ_i and not by the number of processors in the system. Each thread can execute only one subtask at a time, ensuring the limit for maximum parallel execution of a task τ_i to μ_i even if subtasks are ready

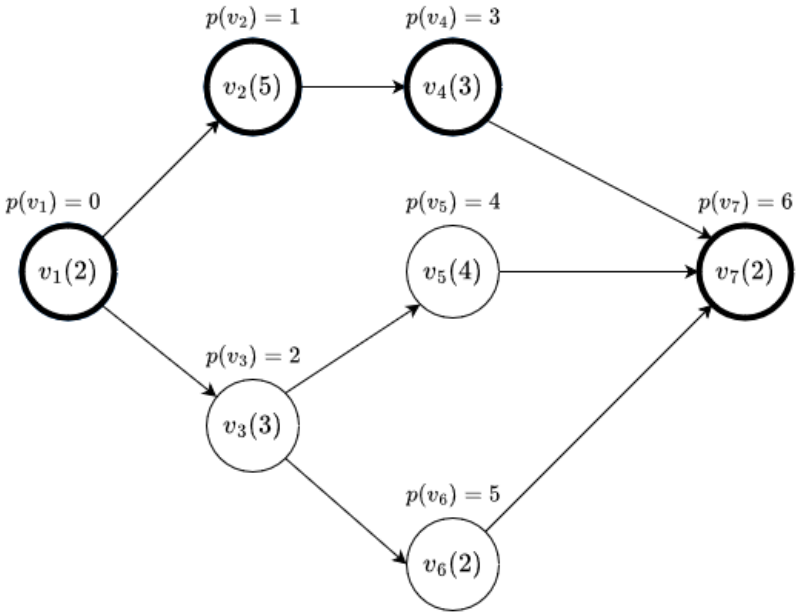


Figure 4.1.: Example DAG

to execute with idle processors [51].

Definition 4.2.1 (Path). *For a given DAG task τ_k, a path $\lambda = (v_{source}, v_1, ..., v_{sink})$ is a sequence of subtasks $v_j \in V_i$ such that v_{source}, v_{sink} represents the source and sink of the DAG graph respectively and $\forall v_j \in \lambda \backslash \{v_{sink}\}, (v_j, v_{j+1}) \in E_i$ [16].*

Informally, a path λ is a sequence of subtasks from the source to the sink of a DAG graph in which there is a precedence constraint between any two adjacent subtasks in λ. Thus, there is no concurrency among subtasks that belong to the same path λ.

Definition 4.2.2 (Length of a Path). *The length of path λ denoted by $len(\lambda)$ for a task τ_i is the sum of WCET C_j of all subtasks $v_j \in \lambda$, formally can be denoted as shown in Equation 4.1 [16].*

$$len(\lambda) = \sum_{\forall v_j \in \lambda} C_j \tag{4.1}$$

Definition 4.2.3 (Critical Path). *The critical path is the path with the longest execution path, formally can be denoted as shown in Equation 4.2 [51].*

$$L_i = max_{\forall \lambda \in G_i} \{len(\lambda)\} \tag{4.2}$$

In the Figure 4.1 the bold nodes represent the subtasks that belong to the critical path.

Definition 4.2.4 (Worst-case workload). *The worst-case workload W_i is the time needed to execute all subtasks of a task τ_i on a single core platform. It is the sum of the Worst Case Execution Time (WCET) of all subtasks of the task τ_i as shown in Equation 4.3 [51].*

$$W_i = \sum_{\forall v_j \in V_i} C_j \tag{4.3}$$

Table 4.1.: Summary of System Model.

Symbol	Description
m	Number of processors
τ	Task set of a real-time application
n	Total number of tasks in τ
τ_i	DAG i^{th} task
T_i	Period of task τ_i
D_i	Relative Deadline of task τ_i
Φ_i	Thread pool of task τ_i
L_i	Critical Path of task τ_i
G_i	DAG representation of task τ_i
V_i	Set of vertices of graph G_i
E_i	Set of edges of graph G_i
$v_{i,j}$	j^{th} subtask of task τ_i
$C_{i,j}$	Worst Case Execution Time (WCET) of subtask $v_{i,j}$
μ_i	number of threads in the thread pool Φ_i of task τ_i
R_i^{ub}	Upper bound on the worst-case response time of task τ_i
$I_k^*(\Delta)$	Exact critical interference on task τ_k over the interval Δ
$I_k(\Delta)$	Critical interference on task τ_k over the interval Δ
$I_{i,k}(\Delta)$	Critical interference on task τ_k by the task τ_i over the interval Δ
$I_{k,k}(\Delta)$	Critical interference on task τ_k by other subtasks of the same task over the interval Δ

4.2.1. Run-time Behavior

For a given single DAG task τ_k, assuming that the task starts at time 0. A subtask or a node is ready to execute at a certain point only if all its ancestors in the graph have completed their execution, and hence source can execute immediately at time 0.

Example 4.2.1. *Considering the DAG shown in Figure 4.1 and assuming the the number of threads in its thread pool Φ_i is $\mu_i = 3$. the Figure 4.2 shows the execution sequence in the μ_i threads.*

At $time = 0$ the source(v_1) is ready and currently there are all three threads available, so the source(v_1) can execute on any of the three threads. At $time = 2$, as soon as the source(v_1) completes its execution, there are two nodes $[v_2, v_3]$ that are ready to execute as v_2 having a higher priority, and again all three threads are available. As the number of subtasks ready <= the number of available threads, there are no subtasks that have to wait; v_2 and v_3 can start their execution in parallel. As WCET of v_2 is 5 and the WCET of v_3 is 3, v_3 completes its execution sooner. Similarly, when v_3 completes its execution at $time = 5$ there are two subtasks that are ready i.e, $[v_5, v_6]$ and now the number of available threads = 2 as at this point v_2 is still executing on thread 1, however still number of subtask ready <= number of

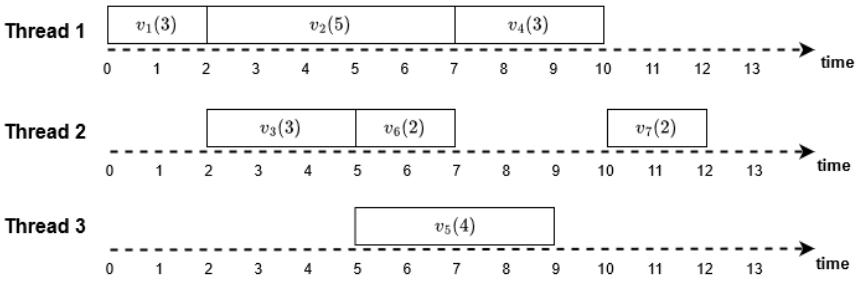


Figure 4.2.: Execution of DAG on three threads

available threads hence v_5 and v_6 start their execution in parallel. At $time = 6$, all the three threads execute the subtasks in parallel. At $time = 7$, v_2 and v_6 complete their execution, the descendants of v_2 are $[v_4]$ which are ready to execute as all of their ancestors have completed their execution. However the children of v_6 i.e, $[v_7]$ is not yet ready as not all of its ancestors have completed their execution, v_7 can only start its execution only when v_4, v_4 and v_6 have completed their execution which happens at $time = 10$, the sink starts its execution at $time = 10$ and completes at $time = 12$. The response time of this DAG = 12.

4.2.2. Response Time Analysis

The work done at [8], [7] and [11] propose interference based response time analysis for the sequential and parallel task models. All the following sections will adopt the same methodology of the thread pool model using DAG tasks [51].

For a sequential task τ_{seq}, the interference is defined as the sum of all intervals of time in which task τ_{seq} is ready to execute but cannot due to higher priority tasks running on the processors. Nevertheless, for a parallel task τ_{par}, the response time of a task τ_{par} is prolonged if the critical path L_i of τ_{par} suffers interference from higher priority tasks as well as other subtasks of the same task τ_{par} [51].

Definition 4.2.5 (Critical interference)**.** *The exact critical interference over an interval Δ on a task τ_k is denoted by $I_k^*(\Delta)$ which is the cumulative time at which the critical path of the task τ_k is ready to execute but cannot execute due to other subtasks of τ_k running on the thread pool or threads of higher priority tasks running on the processors [51][8][11].*

Example 4.2.2. *Considering an example where a task τ_k with threads $\mu_k = 3$ is to be executed on a system with $m = 4$ processors along with other tasks $\tau_i \in \tau$. The interference occurring to the critical path L_k during the interval Δ as shown in Figure 4.3. During the intervals $a, b, and c$, the critical path L_k suffers interference where either other tasks or subtasks that are not of the critical path L_k are scheduled on the processors [51].*

Three different scenarios lead to the interference of the critical path L_k:

1. At interval *a* in Figure 4.3, wherein all the processors are occupied by the other higher priority tasks.

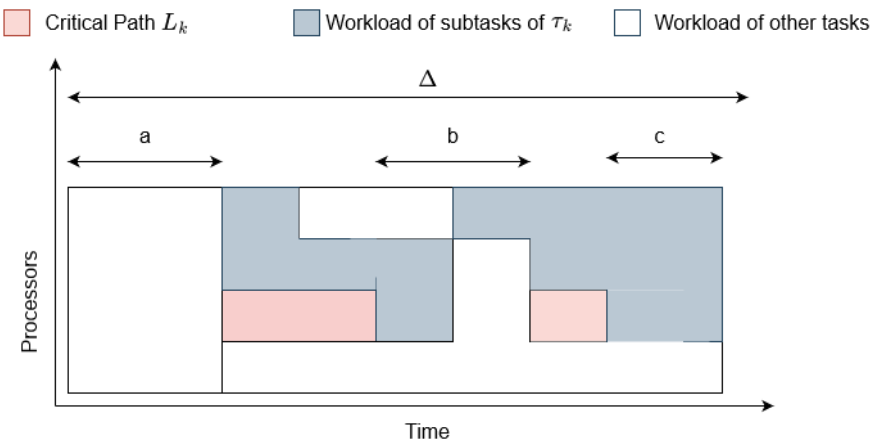


Figure 4.3.: Exact Interference suffered by L_k

2. At interval b in Figure 4.3, where not all the μ_k threads of task τ_k can be scheduled on the processors as the other higher priority tasks occupy them.

3. At interval c in Figure 4.3, where all the μ_k threads of the task τ_k are executing some (or no) threads of higher priority tasks on the processors. However, the tasks of τ_k are executing the subtasks that are not part of the critical path.

The exact critical interference $I_k^*(\Delta)$ for the Example 4.2.2 is the sum of the intervals a, b and c i.e,

$$I_k^*(\Delta) = a + b + c \tag{4.4}$$

As per Definition 4.2.5 and the Example 4.2.2, the critical path L_k of a task τ_k suffers interference from:

1. Other tasks $\tau_i \in \tau$ of the task set, which has a higher priority than that of τ_k running on the processors

2. Subtasks of the same task $\tau_{k,j} \in \tau_k$ that are not part of the critical path L_k

Leading to the following definitions:

Definition 4.2.6. *The critical interference $I_{i,k}(\Delta)$ imposed by task τ_i on the task τ_k over any interval Δ on the critical path is defined as the cumulative workload executed by the threads of task τ_i. While the critical path L_k is ready to execute but cannot execute [51].*

Definition 4.2.7. *The critical interference $I_{k,k}(\Delta)$ imposed by subtasks of task τ_k over any interval Δ on the critical path is defined as the cumulative workload executed by the subtasks that are not part of the critical path of task τ_k while the critical path L_k is ready to execute but cannot execute [51].*

As per Definition 4.2.6 $I_{i,k}(\Delta)$ indicates the total workload of the task τ_i interfering with the task τ_k, for understanding and calculating the interference that is imposed on the task τ_k by each thread of τ_i, the concept of at least p-depth critical interference is used [51].

Definition 4.2.8. *The at least p-depth critical interference $I^p_{i,k}(\Delta)$ imposed by task τ_i on the task τ_k over any interval Δ is defined as the cumulative workload executed by the threads of task τ_i while the critical path L_k is ready to execute but cannot execute while there are at least p threads of task τ_i are simultaneously running on the processors [51].*

As per definitions Definition 4.2.6 and Definition 4.2.8, the critical interference $I_{i,k}(\Delta)$ can be calculated using the p-depth critical interference during an interval Δ as shown in Equation 4.5 [43] [51].

$$I_{i,k}(\Delta) = \sum_{p=1}^{m} I^p_{i,k}(\Delta) \tag{4.5}$$

The work done at [51] proves that the critical interference $I^*_k(\Delta)$ can be upper bounded pessimistically considering the interference caused by other tasks, i.e., Inter-task interference and the interference caused by subtasks of the same tasks, i.e., Intra-task interference separately.

Definition 4.2.9 (Inter-task interference)**.** *The critical inter-task interference $I^\tau_k(\Delta)$ is defined as the interval when the interfering workload of all tasks is distributed evenly among the processors as shown in Equation 4.6 [51]*

$$I^\tau_k(\Delta) = \frac{1}{m} \sum_{\forall \tau_i} I_{i,k}(\Delta) = \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} I^p_{i,k}(\Delta) \tag{4.6}$$

Definition 4.2.10 (Intra-task interference)**.** *The critical intra-task interference $I^v_k(\Delta)$ is defined as the interval when the interfering workload of all interfering subtasks is distributed evenly among the thread pool of τ_k as shown in Equation 4.7 [51].*

$$I^v_k(\Delta) = \frac{1}{\mu_k} I_{k,k}(\Delta) \tag{4.7}$$

The total interference suffered by a task τ_k can be calculated using Equation 4.8, where $I^\tau_k(\Delta)$ represents the Inter-task interference and $I^v_k(\Delta)$ represents the Intra-task interference [51].

$$I_k(\Delta) = I^\tau_k(\Delta) + I^v_k(\Delta) \tag{4.8}$$

$$I_k(\Delta) = \frac{1}{m} \sum_{\forall \tau_i} \sum_{p=1}^{m} I^p_{i,k}(\Delta) + \frac{1}{\mu_k} I_{k,k}(\Delta) \tag{4.9}$$

Using the equation Equation 4.9, an upper bound for the response can be calculated using the Equation 4.10. Conceptually, the response time in the best case would be when the

critical path suffers no interference, i.e., the length of the critical path itself. However, when interference is involved, the response time can be upper bounded pessimistically considering the inter-task interference and intra-task interference separately, as shown in Equation 4.10 [51].

Definition 4.2.11 (Response Time)**.** *The response time R_i of a task τ_i is the time interval between the activation and termination of an instance of the task [21] [51].*

$$R_i = L_i + I_k(\Delta) \tag{4.10}$$

$$R_i = max_{\forall \lambda \in G_i}\{len(\lambda)\} + \frac{1}{m}\sum_{\forall \tau_i}\sum_{p=1}^{m} I_{i,k}^{p}(\Delta) + \frac{1}{\mu_k} I_{k,k}(\Delta) \tag{4.11}$$

Definition 4.2.12 (Worst Case Response Time)**.** *The worst-case response time R_i^+ of a task τ_i is the longest possible response time the task may experience. Formally, it can be denoted as shown in Equation 4.12 [21].*

$$\forall n \in \mathbb{N}^+ : R_i^+ \geq R_i \tag{4.12}$$

5. Event Driven DAG

Real-time applications comprise periodic and sporadic tasks for their responsiveness and determinism. Periodic tasks are generally used for process control, for instance, altitude control in space systems. In contrast, sporadic tasks provide fast responses to external events, for instance, some data-flow applications such as events in Tasking Framework [44]. In periodic tasks, two consecutive activations are separated using a fixed time interval, i.e., Period (T_i), whereas sporadic tasks have a non-determinism behavior, where two consecutive activations are separated by a minimum time interval (T_i) [14].

5.1. Motivation

Algorithms have been presented to handle periodic and sporadic tasks [28] [55] [40] and to compute the response time of systems comprising of sporadic tasks [27]. The main importance of using event-driven applications with a combination of periodic and sporadic tasks is due to the high responsiveness and low resource consumption.

5.1.1. Event-triggered vs Time-triggered

Event-triggered and time-triggered are two fundamental principles used for controlling real-time systems' tasks. For event-triggered systems, tasks are activated in response to relevant events external to the system. For example, when an event in the outside world is detected by a sensor which then causes the activation of a real-time task. Event-triggered systems allow for a faster response at low load but more overhead and chance of failure at high load, which is most suitable for dynamic environments, where dynamic tasks can arrive at any time.

In time-triggered systems, all tasks are carried out at certain times, known as priori. The advantage of using a time-triggered system is the predictable behavior of the system. Time-triggered systems have the opposite properties and are suitable in static environments where most system behavior is known in advance.

Studies prove that event-triggered methods can be combined with time-triggered systems to efficiently include sporadic tasks to allow for high responsiveness and low resource consumption [27] [14].

5.1.2. Sporadic Tasks

Sporadic tasks handle events that are activated at arbitrary points in time but with defined maximum frequency. They are invoked repeatedly with a (non-zero) lower bound - period (T_i) on the duration between consecutive occurrences of the same event. Each sporadic task is invoked repeatedly with a lower bound on the interval between consecutive invocations, i.e., the minimum inter-arrival time between two consecutive activations, known as the Period T_i.

Definition 5.1.1 (Sporadic Task)**.** *A sporadic task is a task that is event-driven i.e, activated by events that occur at an unknown time such that $\delta^+(2) = +\infty$ [21].*

5.2. Arrival Curves

To determine the upper bound of the response time for a task set τ containing multiple sporadic DAG tasks, the inter-task interference calculated pessimistically in [51], models the sporadic task as periodic using the minimum inter-arrival time between two consecutive activations as Period. This section introduces a method that computes the response time less pessimistically than proposed in [51].

Definition 5.2.1 (Activation Trace)**.** *An activation trace $\aleph_i$ of a task τ_k is defined as the absolute time at which an instance of the task τ_k starts its execution. Formally, it can be defined as a function as shown in Equation 5.1. $\aleph_i(n) = t$ indicates that the $n-th$ instance of the task τ_k begins its execution at time t [21].*

$$\aleph_i : \mathbb{N}^+ \longrightarrow \mathbb{R}^+ \tag{5.1}$$

Definition 5.2.2 (Termination Trace)**.** *A termination trace $\beth_i$ of a task τ_k is defined as the absolute time at which an instance of the task τ_k completes its execution. Formally, it can be defined as a function as shown in Equation 5.2. $\beth_i(n) = t$ indicates that the $n-th$ instance of the task τ_k completed its execution at time t [21].*

$$\beth_i : \mathbb{N}^+ \longrightarrow \mathbb{R}^+ \tag{5.2}$$

Definition 5.2.3 (Arrival Curves)**.** *The maximum and minimum arrival curves $\eta_i^+(\Delta t)$ and $\eta_i^-(\Delta t)$ of a task τ_i are functions $\mathbb{R}^+ \longrightarrow \mathbb{N}^+$ that indicates the maximum and minimum number of events that occur during any half-open interval $[t, t+\Delta t), \eta^+(\Delta t)(\eta^-(\Delta t))$ [21].*

Informally, the function $\eta_i^+(\Delta t)$ over an interval Δ returns the maximum number of activations that occur in the interval Δ of task τ_i, whereas the function $\eta_i^-(\Delta t)$ over an interval Δ returns the minimum number of activations that occur in the interval Δ.

$$\eta_i^+ : \Delta T \longrightarrow n \tag{5.3}$$

The Figure 5.1 shows that, the maximum number of activations in over a time interval of 1 is 2 i.e, $\eta_i^+(1) = 2$, similarly the maximum number of activations over a time interval of 2

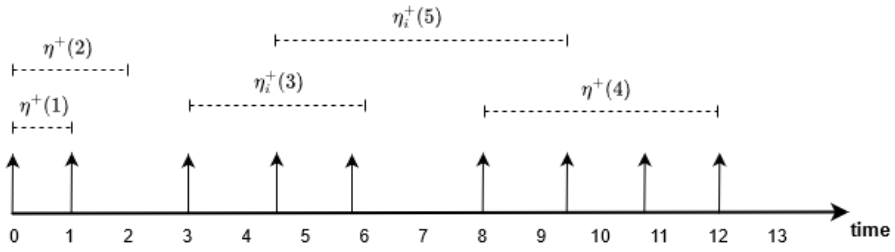


Figure 5.1.: The maximum arrival function η_i^+ on an activation trace

is also 2 i.e, $\eta_i^+(2) = 2$. Also, $\eta_i^+(3) = 3$, $\eta_i^+(4) = 4$ and $\eta_i^+(5) = 4$. The Figure 5.2, shows a summary of the arrival curves output on the activation trace as shown in Figure 5.1 which shows that the Arrival functions are non-decreasing and are sub-additive [36][21].

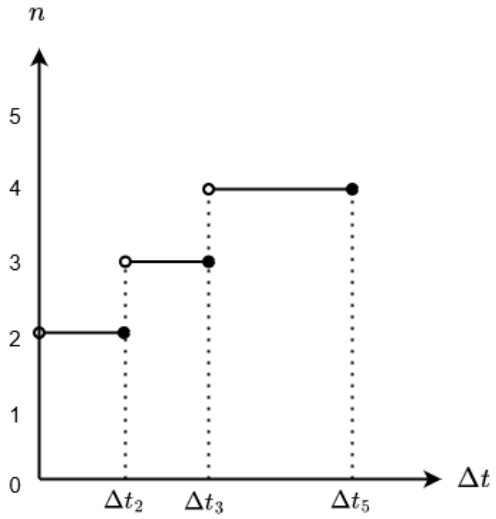


Figure 5.2.: The maximum arrival function η_i^+

Definition 5.2.4 (Distance Functions)**.** *The maximum and minimum distance functions $\delta_i^+(n)$ and $\delta_i^-(n)$ of a task τ_i are functions $\mathbb{N}^+ \longrightarrow \mathbb{R}^+$ that indicates the maximum and minimum time intervals during at which at most n events occur [21].*

$$\delta_i^+ : n \longrightarrow \Delta T \tag{5.4}$$

Informally, the function $\delta_i^+(n)$ for a given number of activations of a task τ_i returns the maximum time interval (distance) for n activations, whereas the function $\delta_i^-(n)$ for a given number of activations of a task τ_i returns the minimum time interval (distance) for n activations.

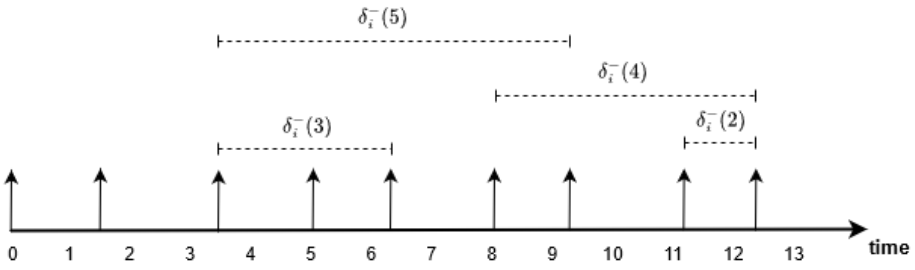


Figure 5.3.: The minimum distance function δ_i^- on an activation trace

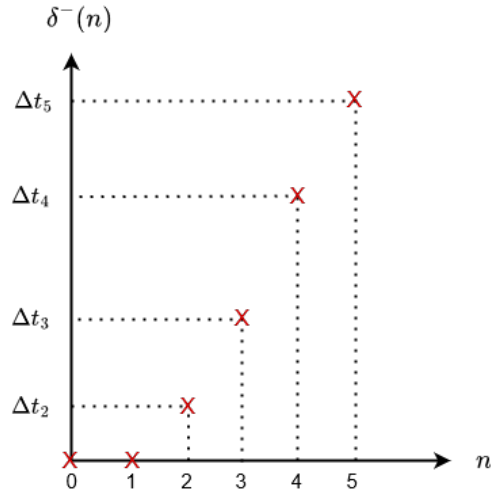


Figure 5.4.: The minimum distance function δ_i^+

The Figure 5.3 shows that, the minimum time required for 2 activations is 1 i.e, $\delta_i^-(2) = 1$, similarly the minimum time required for 3 activations is 3 i.e, $\delta_i^-(3) = 3$. Also, $\delta_i^-(4) = 4$ and $\eta_i^+(5) = 5$. The Figure 5.4 shows a summary of the minimum distance outputs for the interval shown in Figure 5.3, which shows that minimum distance functions are non-decreasing and super-additive.

5.2.1. Response Time Analysis

Definition 5.2.5 (Simplified p-depth workload distribution)**.** *The Simplified p-depth workload distribution $\mathcal{W}_i^p(\Delta)$ over any interval Δ is defined as the workload the p^th thread of the task τ_i executes when the complete workload of the task is distributed evenly among all the threads μ_i in the thread pool Φ_i in the interval Δ, can be denoted using the Equation 5.5 [51].*

$$\mathcal{W}_i^p(\Delta) = \left(\left\lfloor \frac{\Delta + R_i^{ub} - L_i}{T_i} + 1 \right\rfloor \right) \frac{W_i}{\mu_i} \tag{5.5}$$

The work done at [51], uses the p-depth workload distribution to calculate the inter-task interference, which is computed using the Equation 5.5, the equation computes the p-depth workload distribution pessimistically using the Period T_i for sporadic tasks, the usage of arrival curves allows computing the response time less pessimistically by not effecting any assumptions made in the model and does not break any definitions. The value inside the parentheses in Equation 5.5 represents the number of task releases in Δ, which is calculated using the Period T_i, to compute the number of activations in an interval Δ, the arrival curves calculates the maximum number of activations over the given interval. Using this, a less pessimistic response time is computed, where the interval passed to the arrival curve is the same as that of the Equation 5.5 as depicted in Equation 5.6.

$$\mathcal{W}_i^p(\Delta) = \eta_i^+(\Delta + R_i^{ub} - L_i)\frac{W_i}{\mu_i} \tag{5.6}$$

5.3. Synthetic Test Case Generation

For our experiments, sporadic synthetic tasks are generated, this section shows the proposed algorithm to generate them. The algorithm generates a random trace simulating an activation trace of the sporadic task, and the following conditions are to be followed:

1. The minimum distance between two activations is the Period T_i as shown in Equation 5.7.

$$\delta_i^-(2) = T_i \tag{5.7}$$

2. The maximum distance between two activations is twice the Period, i.e., $2 * T_i$.

3. The absolute time for an activation trace $\delta_i^-(n)$ is computed in the range shown in Equation 5.8 i.e., the minimum distance required for n activations is the sum of the minimum distance required for $n-1$ activations and the Period, whereas the maximum distance required for n activations is the sum of minimum distance required for $n-1$ activations and two times the Period as shown in Equation 5.8.

$$\delta_i^-(n) \geq \delta_i^-(n-1) + T_i \leq \delta_i^-(n-1) + (2 * T_i) \tag{5.8}$$

4. The above conditions ensure that the maximum distance required for n activations is less than or equal to two times the minimum distance required for n activations as shown in Equation 5.9.

$$\delta_i^+(n) \leq 2 * \delta_i^-(n) \tag{5.9}$$

5.3.1. Algorithm

1. A trace of length hundred is created, where the value of an index of the trace represents the absolute time at which the an instance of the task is activated.

2. The first value of the trace being zero i.e, $trace[0] = 0$ which indicates that the first activation of the task is at the $time = 0$

3. The second value being the Period T_i itself i.e, $trace[1] = T_i$, allowing to ensure that the minimum distance between two activations is the period T_i.

4. The following trace elements are computed using the following steps:

   a) As the minimum distance between two activations is the Period T_i. The new trace element is computed using the previous trace element, the Period, and a random uniformly distributed value in the range (0, 1), ensuring that the next activations are in the range:

   $$(trace[i-1] + T_i, trace[i-1] + (2 * T_i)) \tag{5.10}$$

   b) Which still ensures that the minimum distance between two activations is the Period T_i and the maximum distance between two activations is $2 * T_i$ as shown in Listing 5.1.

Listing 5.1: Compute trace element for index > 1

```
trace.append(trace[i - 1] + random.uniform(0, period) + period)
```

6. Intra-Task Priorities DAG

This chapter provides the proof that the order of execution of nodes of a Directed Acyclic Graph i.e., intra-task priorities, allows for tightening the worst-case upper bound of the intra-task interference, and presents the analysis and an algorithm to compute the worst-case intra-task interference for DAGs with intra-task priorities.

6.1. Background

The current state-of-the-art methods [20] [39] [45] [29] [6] [51] calculates the upper bound response time and intra-task interference of a single DAG task considering work-conversing scheduling algorithms where intra-task priorities i.e, priorities at the subtask level are not assigned. These algorithms execute an arbitrary available node and do not specify the order of execution of the subtasks. The algorithms mentioned in [20] [39] [45] [29] [6] [51] have different strategies to execute the eligible vertex, however the response time bound is the same for all the algorithms [25].

Building upon the work done at [51] where it is proven that for a given DAG Task τ_k executing on multi-core processors, the subtasks are consigned to and executed by the worker threads of a thread pool with μ_k threads. The response time and intra-task interference for a single task τ_k can be upper bounded by equations Equation Classical Upper Bound Function and Equation 6.1 respectively.

$$R_k^+ <= L_k + \left(\frac{W_k - L_k}{\mu_k}\right) \qquad \text{(Classical Upper Bound Function)}$$

$$R_k^{ub} \leftarrow \frac{1}{\mu_k}(W_k - L_k) \tag{6.1}$$

The methods [20] [39] [45] [29] [6] and [51] do not assign priorities, thus the execution order of nodes are unpredictable as there is no way to specify the order in which the execution should take place, it is scheduled by the OS non-deterministically and hence the upper bound in Equation Classical Upper Bound Function is valid for all DAGs. Therefore conceptually the equation Equation 6.1 and Equation Classical Upper Bound Function can be considered as a safe upper bound.

6.2. Motivational Example

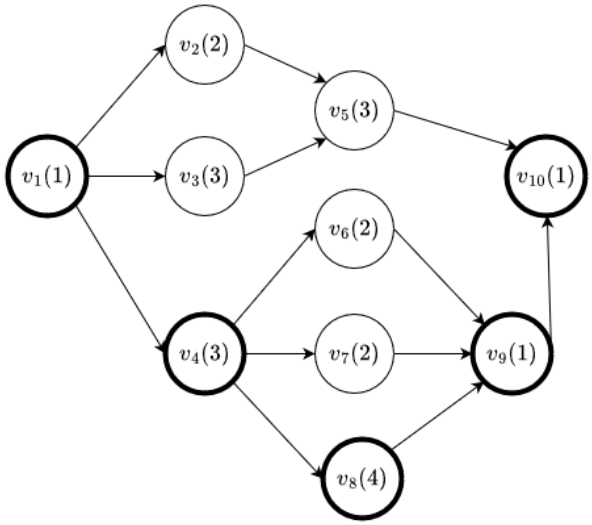


Figure 6.1.: A DAG task

This section illustrates the reasons with an example that shows a study is required on computing end-to-end response time for DAGs real-time task running on a multi-core processor with intra-task priorities. Firstly, for DAG tasks with priorities assigned to their subtasks, it can be proven that the end-to-end response time can be reduced compared to the DAGs where intra-task priorities are not assigned. Secondly, there are real-world, real-time applications scenarios where priorities are predetermined and assigned; for instance, in parallel frameworks, it is feasible to use the priority clause to state the priority of a task construct [24].

Scheduling of a DAG task τ_k there can be instances where the number of the eligible subtasks are larger than the number of worker threads available μ_k in the thread pool, in the algorithms where priorities are not assigned, an arbitrary subtask is chosen to execute, in the following, it is proven that with a proper order of execution the response times can be reduced as and allows to improve predictability of how the subtasks can be scheduled on the threads. For the DAG as shown in the Figure 6.1, assuming that the task is being scheduled on a thread pool with two threads, the response time computed when intra-task priorities are not assigned, calculated using the Equation Classical Upper Bound Function [51] gives a total 16 as shown in Equation 6.2.

$$L_k + \left(\frac{W_k - L_k}{\mu_k}\right) = 10 + \left(\frac{22 - 10}{2}\right) = 16 \tag{6.2}$$

However, there is indeed multiple possible order of executions as shown in Figure 6.2, Figure 6.3 and Figure 6.4 when intra-task priorities are assigned; the figures prove that the response time can be shorter than the calculated value using Equation 6.2.

In the Figure 6.2 the total response time is 14. Firstly, only the source is added to the priority queue, as at the beginning, both the threads are available the source begins its execution in any one of the threads; once the source has completed its execution, we add the children of the source, there are three available nodes, i.e., $[v_2, v_3, v_4]$ that is added to the priority queue in this particular order, at this iteration there are three available nodes and two available threads, v_2 starts its execution on Thread 1 and v_3 on Thread 2. v_4 cannot start its execution as there are no available threads, v_5 must wait for v_2 or v_3 to complete its execution, as v_2 completes its execution first, we can see if the children of v_2 can be added to the priority queue, however v_5 cannot be added at this point, as v_5 is dependent on both v_2 and v_3, as v_4 is alone at this point in the priority queue, so v_4 starts its execution right after v_2 has completed its execution. After v_3 completes its execution, all the dependencies of v_5 have completed their execution; now v_5 is added to the priority queue. As currently, only Thread 2 thread is available, v_5 is executed on Thread 2. During the execution of v_5, v_4 has completed its execution, and all its children $[v_6, v_7, v_8]$ are added to the priority queue in the given order. Furthermore, at this stage, only one thread, Thread 1, is available; there are currently three available nodes, and out of those, v_6 is chosen for execution. During this execution, v_5 completes its execution, but its children v_{10} cannot be added as there is another dependency v_9 that has not been executed yet. So after v_5 completes, there are currently two nodes in the priority queue, which are $[v_7, v_8]$, v_7 is popped and starts its execution on Thread 2. Once v_6 completes its execution, v_9 cannot be added as two dependencies are not executed. Hence v_8 starts its execution on Thread 1 during v_8 execution, v_7 completes the execution, however as of at this stage v_9 cannot be added until v_8 also completes. So at this point, the thread is available, but there is no execution. After v_8 completes, now v_9 can be added to the priority queue, only v_9 completes, then the sink v_{10} is added to the priority queue and executed.

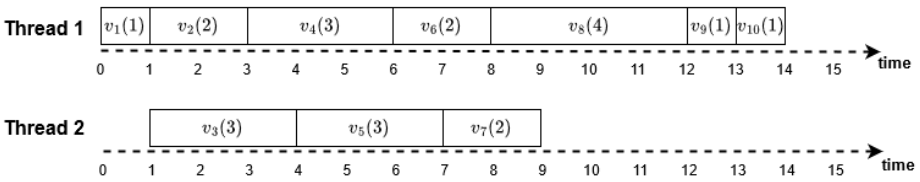


Figure 6.2.: Execution Scenario 1

In the Figure 6.3, the total response time is 13. Like the previous execution, the source starts its execution; when the source completes, its children are added to the priority queue in the order $[v_4, v_3, v_2]$, which is different from the previous order of execution. At this stage, there are two available threads and three available nodes; as v_4 is at the top of the queue, v_4 begins its execution on Thread 1 in parallel to v_3, which begins its execution on Thread 2. As in this case, both complete their execution simultaneously; now, the children of v_4 and v_3 can be checked if they are ready to execute. The children of v_4 are added to priority queue in the order of $[v_6, v_8, v7]$ now the priority queue is $[v_2, v_6, v_8, v7]$ the children of v_3 are just v_5 however v_5 cannot be added as one of its ancestor v_2 has not yet been executed. As there are two available threads, v_2 and v_6 start their execution in parallel on Thread 1 and Thread

2, respectively. Again v_2 and v_6 complete their execution at the same time, now v_5 can be added to the priority queue in the order $[v_8, v_5, v_7]$. But, v_9 i.e, children of v_6 cannot be added until v_7 and v_8 have been executed. At this stage the priority queue is $[v_8, v_5, v_7]$. Now as there are two available threads; v_8 and v_5 start their execution in parallel, v_5 completes its execution first, however children of v_5 which is v_{10} cannot be added to the priority queue, as v_{10} ancestors are both v_5 and v_9. Now the priority queue is just $[v_7]$, so v_7 starts its execution right after v_5. During the execution of v_5, v_8 completes its execution; however still v_9 cannot be added as v_7 has not yet been executed. Hence this thread Thread 1 stays idle. After v_7 completes its execution now, v_9 can be added to the priority queue, as there are two available threads, v_9 can execute on either, and only after v_9 completes its execution, now all the ancestors of the sink have completed their execution, hence sink is added to the priority queue and executed.

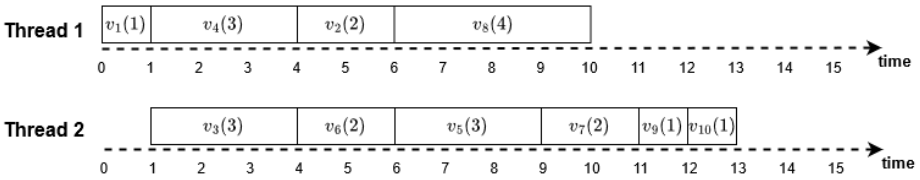


Figure 6.3.: Execution Scenario 2

Similarly, another order has the least response time in the last two execution orders, as shown in Figure 6.4. From this example, we can see that the order of execution of subtasks affects the response time. The response time when priorities were not assigned gave a value of 16 as shown in Equation 6.2. In the above figures, we got the least response time of 12 in Figure 6.4. Conceptually, the response time can be tightened when the order of execution is managed, which is possible when priorities are assigned to the subtasks.

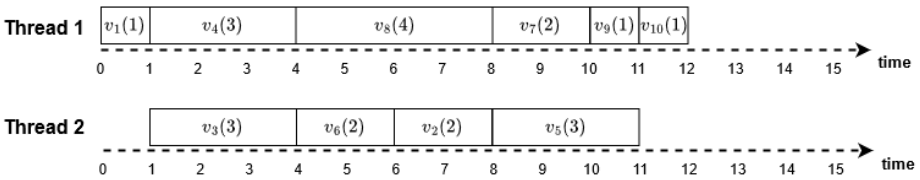


Figure 6.4.: Execution Scenario 3

6.3. Compute Intra-Task Interference Bound

The above section proves that there is a necessity for an algorithm to compute the intra-task interference of a DAG tasks when priorities are assigned to the subtask. This section introduces an algorithm that computes the intra-task interference for a given DAG task. The algorithm computes the intra-task interference assuming that the priorities of the nodes do not conflict with the topology order of the graph, i.e., a node's priority is not higher than any of its descendants, and the algorithm can only be used for DAGs with a single source and sink with intra-task priorities assigned. Without priorities, the algorithm should not be used to calculate the upper bound for a DAGs. Furthermore, for a given task where priority is not assigned to the subtasks, in this scenario, topological priority assignment is used by default. There are exponentially many possible priority assignments complying with the above assumption [25]. Without loss of generality, the algorithm assumes that the whole DAG is released at time 0.

6.3.1. Algorithm

The algorithm mainly depends on two Classes - Thread and Job; their class definition is shown in Listing 6.2 and Listing 6.1 respectively. In the class Thread, *thread_id* is a unique identifier, and the attribute *time* represents the total absolute time of execution of tasks or idle time of threads at any point in time. In the class Job, *accumulate* is the field that represents the absolute time at which the subtask is terminated on the thread. The algorithms required the DagTask, for which the intra-task interference has to be calculated as an input. Following are the steps that are used to compute the intra-task interference.

1. The algorithm creates a min heap queue[18] of the size of the number of threads of the given DAG task. Each node in a heap stores a thread object; see Listing 6.2.
2. Creates a list *parallelism* which indicates a set of available nodes that can run in parallel at any given time during the execution of the task. As the algorithm assumes a single source and a single sink DAG graph, the source is added first to the *parallelism* list.
3. Until the length of the *parallelism* list is not zero; the following steps are executed:
   a) The node with the highest priority is popped from the parallelism list.
   b) For a given node pushes the node to the threads using the following steps:
      - If there are no ancestors for the given node, this is the case where the node can be executed in any of the threads at any time, as there are no ancestors. Hence the algorithm chooses the thread with minimum time and executes the node on this thread.
      - If there are ancestors, collect all the ancestors of the node to ensure that all the ancestors have completed their execution; if any of the ancestors has not yet been executed, the algorithm does not add the node to the threads. If all the node ancestors have been visited, then find the ancestor node with the

maximum *accumulate* named *max_accumulate_node*, which indicates that the node can start its execution only after the *accumulate* of *max_accumulate_node*. In the next step, the algorithm chooses a thread that satisfies one of the following conditions: Firstly, if the thread time is greater than the *accumulate* of *max_accumulate_node* - which indicates that all its ancestors have completed their execution. Now the node can be executed on this thread. Secondly, suppose the sum of the thread time and the *WCET* of *max_accumulate_node* is greater than or equal to that of the accumulate of *max_accumulate_node*. In that case, the node is to be executed on a different thread than the thread where its ancestor *max_accumulate_node* was executed; this condition ensures that the node can start its execution here as the *max_accumulate_node* has completed its execution in one of the other threads.

- After the node is pushed to a thread, update the thread times by including in the *WCET* of the node and also update the node *accumulate* to the thread time and the node *thread_id* to the thread where it was executed.

c) Once the node is pushed to the threads, check if the length of *parallelism* is zero; if it is zero, this condition verifies that there exists a layer in the DAG. Now the algorithm updates the threads' time with the maximum time of the threads. In other words, a buffer is created in the threads where the layer was not executed.

d) Now, the algorithm collects the next layer and is added to the parallelism.

e) For each node in the new layer, the node is checked if all its ancestors are visited; only then can a node be executed. If not, the node is still added to parallelism but is not pushed to the threads until and unless all its ancestors have completed their execution.

4. Finally, the maximum time of the threads represents the sum of intra-task interference and the critical path.

5. The algorithm then computes the intra-task interference by subtracting the critical path and returns the intra-task interference.

Listing 6.1: Job Class and its attributes

```
class Job:
    def __init__(self, wcet: int, **kwargs):
        self.wcet = wcet
        self.relative_completion_time = wcet
        self.priority = kwargs.get('priority') if 'priority' in kwargs else 0
        self.visited = False
        self.critical = False
        self.accumulate = 0
        self.thread_id = -1
        self.priority = 0
        self.execute = True

    def __lt__(self, other):
        return self.priority < other.priority
```

Listing 6.2: Thread Class and its attributes

```
class Thread:
    def __init__(self, thread_id):
        self.time = 0
        self.thread_id = thread_id
```

The illustration of scheduling the subtasks with intra-task priorities using the working explained in subsection 6.3.1 is shown for the DAG Figure 6.5 at Example 6.3.1. The detailed implementation of the algorithm to compute intra-task interference is shown at subsection 8.5.2.

Example 6.3.1. *Consider a DAG task as shown in Figure 6.5 with priorities assigned as shown in the figure, which is executing on a thread pool with $\mu_i = 2$ threads in the thread pool. The intra-task interference is calculated using the steps shown in subsection 6.3.1.*

The following diagrams show the iterations to compute the response time using the algorithm mentioned in subsection 6.3.1. The buckets show a detailed stack of how the nodes are added to the threads, which simulates exactly as executing on a thread. The heap on the right represents the thread's absolute time at any point.

The value in the threads represents an end-to-end thread time which is the sum of critical path and intra-task interference. In the figures Figure 6.5, Figure 6.6 and Figure 6.7 the bold components are the nodes of the critical path, for the DAG shown in Figure 6.5 the critical path is ten. After the execution of the algorithm 17 for the DAG shown in Figure 6.5 the maximum end-to-end thread time as shown in the figures Figure 6.6 and Figure 6.7 is fourteen. Hence the intra-task interference is calculated using the following computation,

intra-task-interference = max-thread-time - critical-path

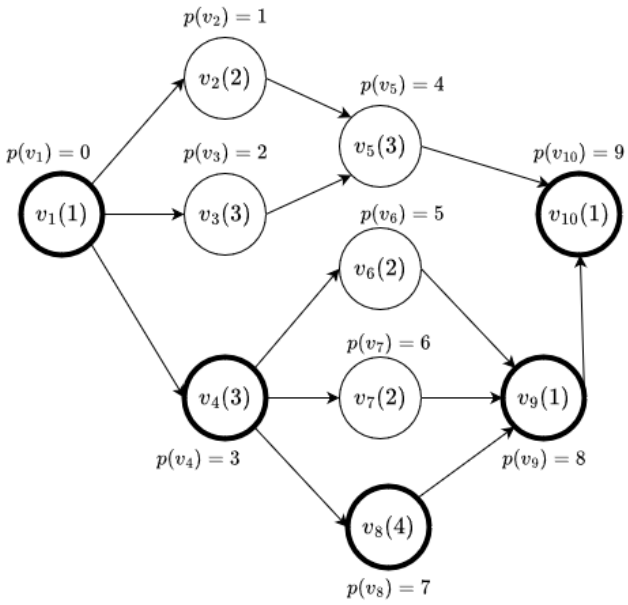


Figure 6.5.: Example illustrating intra-task interference computation

using the computation the intra-task interference for the DAG shown in Figure 6.5 is

intra-task-interference = 14 - 10 = 4

Whereas the intra-task priority when using the Equation 6.1 when intra-task priorities are not assigned [51] the intra-task interference is calculated as

$$\frac{22 - 10}{2} = \frac{12}{2} = 6 \tag{6.3}$$

Wherein twenty-two is the workload of the DAG shown in Figure 6.5, and the two represent the number of threads used.

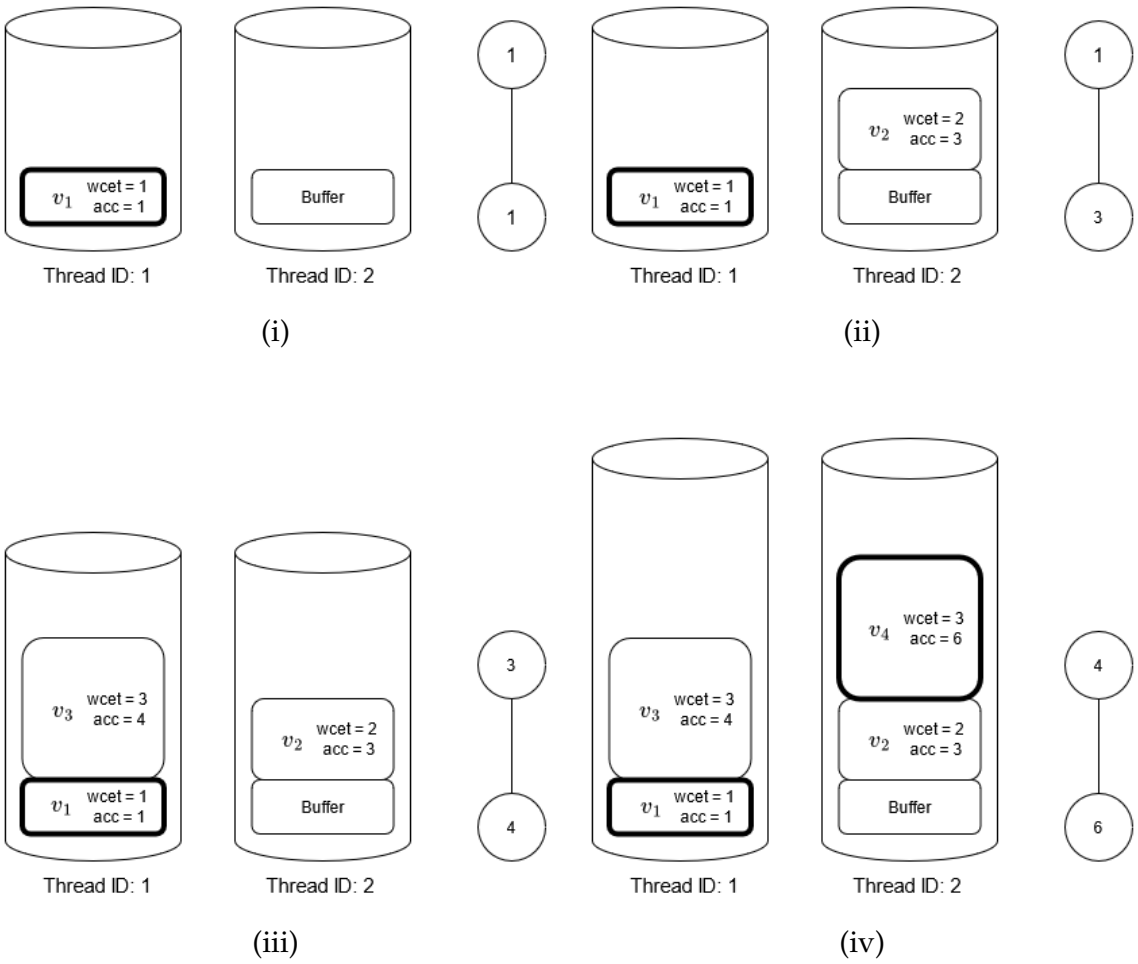
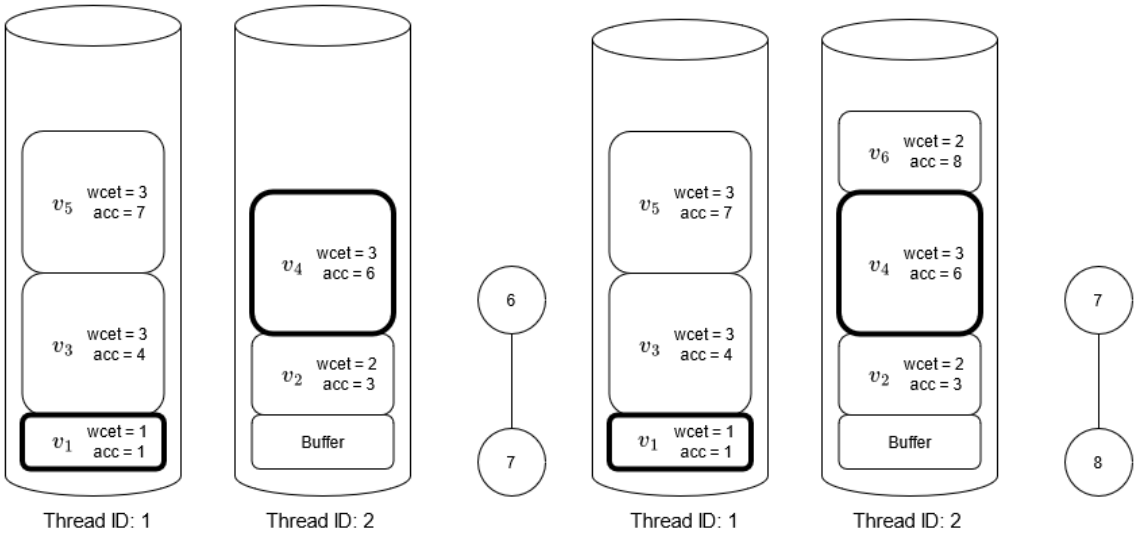
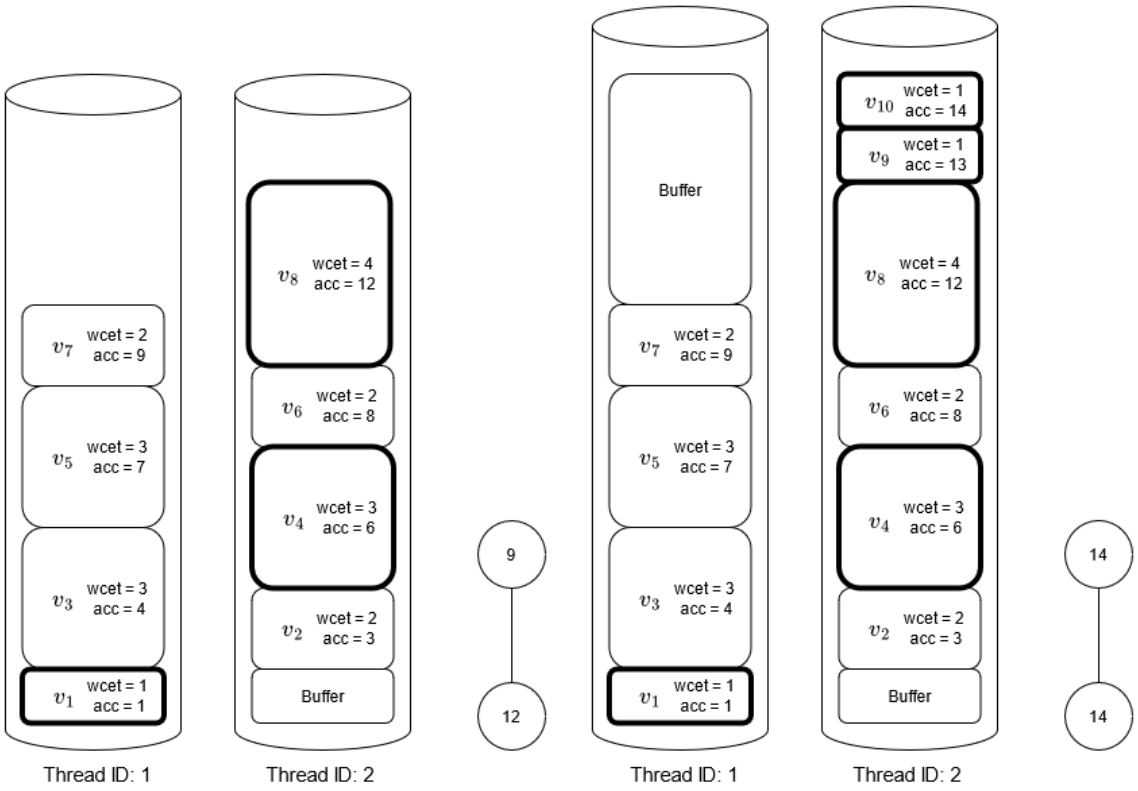


Figure 6.6.: Intra-task interference computation Iterations (i) - (iv)



(v) (vi)



(vii) (viii)

Figure 6.7.: Intra-task interference computation Iterations (v) - (viii)

7. Priority Assignment DAG

As discussed in the chapter 6, it is proved that for a given task τ_k executed on a thread pool of μ_k threads, the response time is affected based on the order of execution of subtasks, i.e., different priority assignment results in different response time bounds. Using a priority assignment algorithm ensures a control in the execution order of the subtasks and allows for tightening the intra-task interference bound. In this section, the most common algorithms are explained that assign priorities based on the characteristics of the graph. The only assumption is that the priority order of nodes does not conflict with the topological order of the Directed Acyclic Graph (DAG), i.e., a subtask's priority is not higher than any of its descendants.

7.1. Topological Priority Assignment

As the name suggests, the priorities are assigned to the subtasks per the topology order of the DAG. For the DAG as shown in Figure 6.1, the topology order is : $[v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}]$. The priority is assigned to each node sequentially from 0 to n, such that $p(v_1) = 0, p(v_2) = 1 ..... p(v_{10}) = 9$ as shown in Figure 7.1. And as the priorities are assigned based on the topological order of the DAG, it does not break the assumption that a subtask's priority is not higher than any of its descendants. The detailed algorithm for assigning the priorities to a DAG is shown in algorithm 12.

7.2. Random Priority Assignment

The algorithm assigns priorities to the nodes Layer-by-Layer i.e., A layer in a Directed Acyclic Graph (DAG) consist of nodes or subtasks that are ready at the same time in parallel. For instance, for a single source DAG, layer 1 consists of only the source node, layer 2 consists of all descendants of the first layer, and this is followed until the sink.

The algorithm starts from the source with priority $p(source) = 0$. The next layer is collected, i.e., all the descendants of the previous layer, and the priorities are assigned only to the nodes that belong to this layer. This assignment is done for random nodes in a layer but sequential priority, i.e, $p(get\_random\_node(nodes\_of\_layer)) = 1, p(get\_random\_node(nodes\_of\_layer)) = 2 ... p(get\_random\_node(nodes\_of\_layer)) = len(nodes\_of\_layer)$ and after which the next layer is collected and assigns the priorities in the same way. This type of assignment ensures that if Layer A is before Layer B then p(any node at layer B) > p(any node at layer A) hence not breaking the assumption that a subtasks' priority is not higher than any of its descendants.

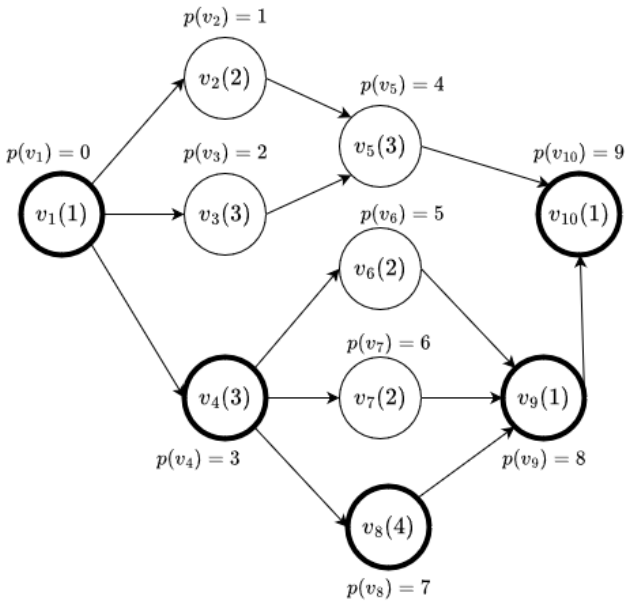


Figure 7.1.: DAG task with Topological Priority Assignment

For the DAG as shown in Figure 6.1, starting with the source v_1 is assigned $p(v_1) = 0$, now all the children of the source is collected denoted as Layer A in Figure 7.2, now priorities are assigned to random nodes of the layer but sequential priority i.e., a random node is chosen between $[v_2, v_3, v_4]$ and assigned a priority of one, for instance $p(v_4) = 1$, this step is followed until priorities are assigned to all the nodes of the Layer A. After this, the next layer is collected, i.e., all the descendants of all the nodes of Layer A, which results in Layer B comprising of nodes - $[v_5, v_6, v_7, v_8]$, now again priorities are assigned only to the nodes of the Layer B. And these steps are followed until the sink is reached.

7.3. WCET Priority Assignment

Similarly as the method Random Priority Assignment explained in section 7.2, the WCET Priority Assignment algorithm also assigns priorities to the nodes Layer-by-Layer. The algorithm starts from the source with priority $p(source) = 0$, then the next layer is collected, and the priorities are assigned only to the nodes that belong to this layer. This assignment is done by sorting the nodes in descending order of a layer based on the WCET of the nodes. Then priorities are assigned sequentially, i.e., for a layer, the node with the highest WCET gets the highest possible priority. And after which, the next layer is collected and assigned the priorities in the same way. This type of assignment ensures that if Layer A is before Layer B then p(any node at layer B) > p(any node at layer A) hence not breaking the assumption that a subtasks' priority is not higher than any of its descendants.

For the DAG as shown in Figure 6.1, starting with the source v_1 is assigned $p(v_1) = 0$,

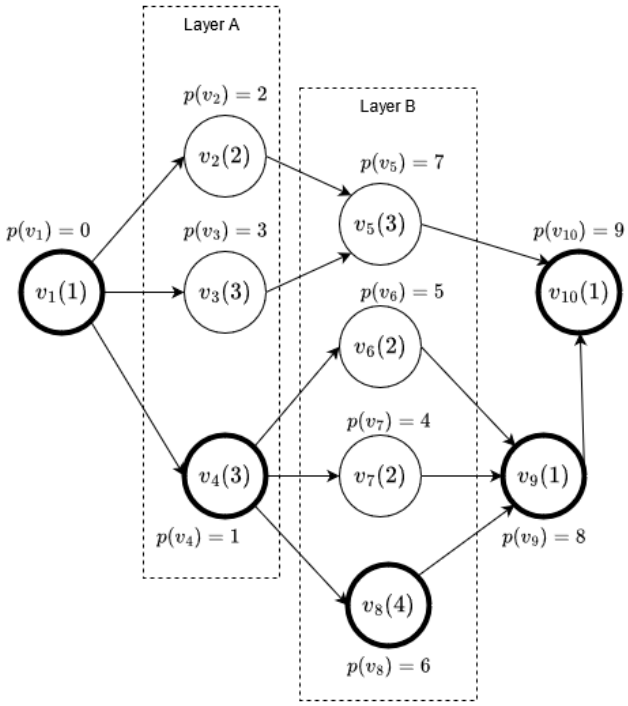


Figure 7.2.: DAG task with Random Priority Assignment

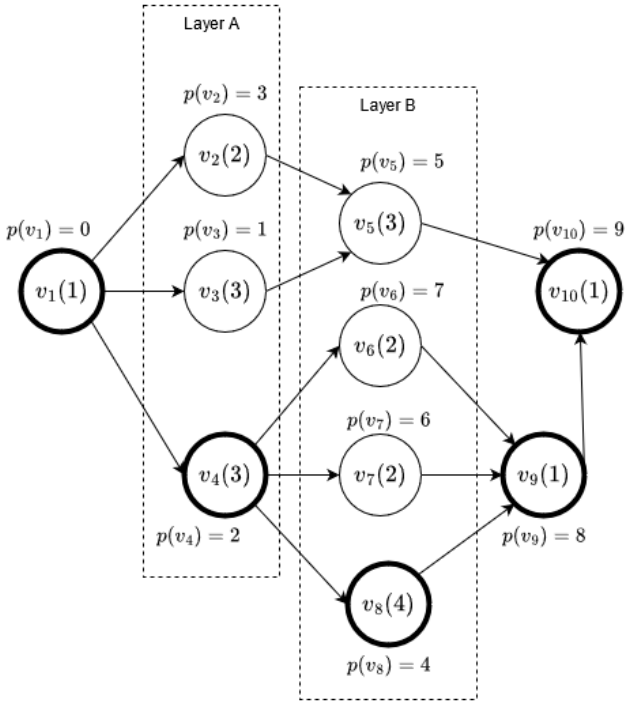


Figure 7.3.: DAG task with WCET Priority Assignment

now all the children of the source is collected denoted as Layer A in Figure 7.3, the nodes in this layer are sorted in descending order based on the WCET, for the layer A the nodes are arranged as - $[v_3, v_4, v_2]$ as v_3 as the highest WCET and v_2 the lowest. Then priorities are assigned to these nodes of the layer A but sequentially i.e., $p(v_3) = 1$ and $p(v_2) = 3$, these steps are followed until priorities are assigned to all the nodes of the Layer A. After this, the next layer is collected, i.e., all the descendants of all the nodes of Layer A, which results in Layer B comprising of nodes - $[v_5, v_6, v_7, v_8]$, now again priorities are assigned only to the nodes of the Layer B. And these steps are followed until the sink is reached.

The difference between Random and WCET Priority Assignment is based on how the nodes of a layer are arranged. In Random Priority Assignment, the nodes of the layer are randomly shuffled, whereas in WCET Priority Assignment, the nodes of the layer are sorted in descending order according to their WCET.

8. Implementation

This chapter elaborates on the implementation of the proposed algorithms in this work. The development, testing and experimentation were all implemented in Python 3.10 [19] and its libraries, along with external packages. Firstly, Networkx[3] for graph manipulation, pyCPA[12] for modeling the activation trace required for arrival curves and NumPy[47] for generation of random gamma and uniform distribution which is required for creating random Directed Acyclic Graph's (DAGs).

The following sections explain the implementation of each module separately and end with a section that combines all the modules.

8.1. System Model

Starting with the implementation that is required for the creation of a Directed Acyclic Graph task. As illustrated in the class diagram at Figure 8.1, *DagTask* extends the base class *Task*, wherein a single *DagTask* consists of a single graph that is represented by the class *Dag*, *period*, *deadline*, *number_of_threads*, *utilization* are the main attributes that are required for the *DagTask*, the methods that involve the delta eta functions are explained in the section 8.2.

Most importantly, this section is about the representation and manipulation of a Directed Acyclic Graph, the implementation uses the package Networkx [3] for the creation of a Directed Graph [4], the methods in the class *Dag* are implemented such that a single *Dag* object consists of n nodes, wherein each node comprises of a single Job (subtask) and the corresponding outgoing edges from the object. These nodes are used in the creation of a directed graph using Networkx [4]. Methods of the *Dag* class, allow to add a node or edge to the graph, which in turn uses the methods provided by Networkx to manipulate the Directed graph, compute the critical path and workload distribution as shown in Listing A.2 and Listing A.3 respectively. The methods *generate_trace_uniform*(), *generate_delta*() are explained in the section 8.2.

An object of the class *DagNode* consists of an instance of Job and an instance of edge. An edge which is an instance of *Edge* represents the outgoing edges from one source to multiple targets which are other *DagNode* instances. In the *Job* class, which is called a subtask, consists of a Worst Case Execution Time which is used in methods that require the scheduling of the *DagNode*, for instance, response time and critical path calculation. *relative_completion_time* is used for calculation of the critical path, *priority* which represents the priority of the subtask in the task, *visited* which is a variable that is used to represent whether a subtask has been visited or not required during the iteration of the graph for instance to get the topological sorted graph, *critical* which indicates whether the subtask is a part of the critical path or not,

accumulate which is used to represent the absolute time of execution in the thread required for response time calculation, *execute* which represents whether a subtask is ready to be executed or not, and finally, *thread_id* whose value represents the *id* of a thread in which the subtask was executed.

8.1.1. Class Diagram

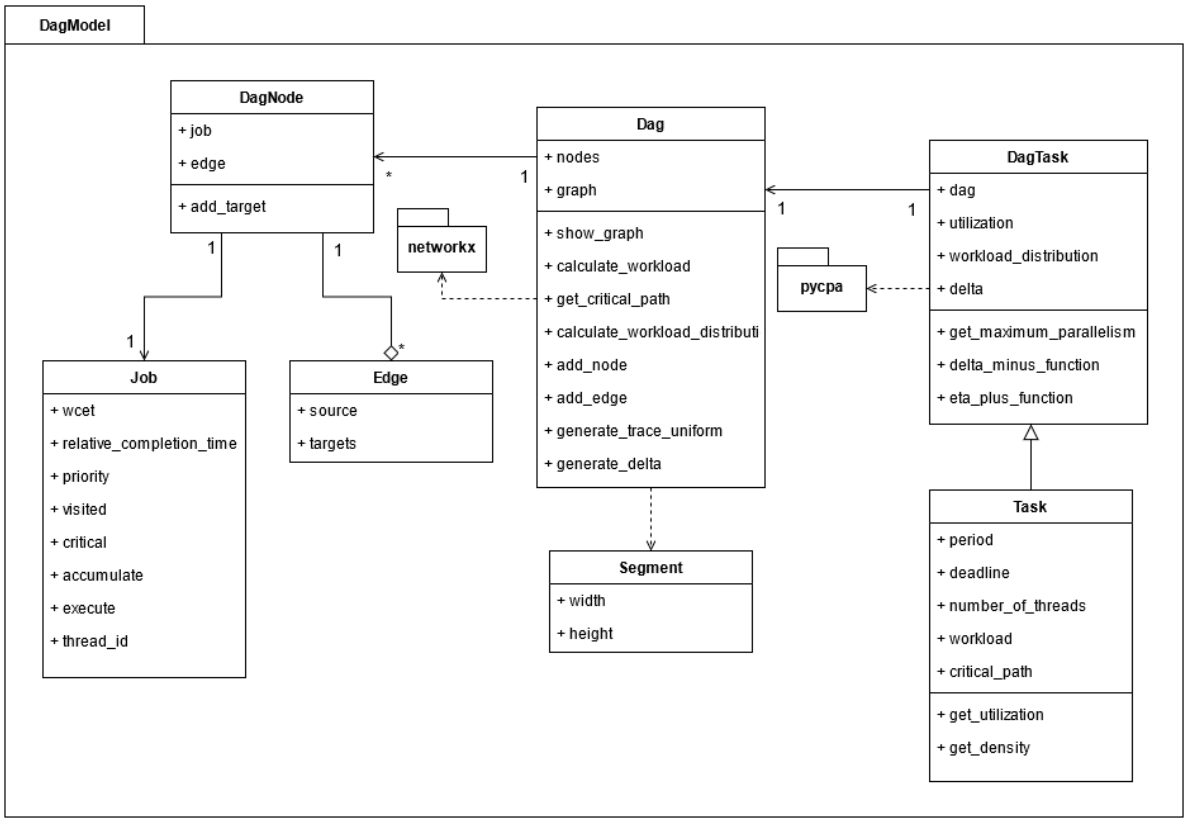


Figure 8.1.: Class Diagram: DagModel

8.2. Event Driven DAG

This section explains the implementation of the usage of arrival curves and the distance function i.e, eta functions η^+ η^- and delta functions δ^+ and δ^- which are required for calculation of the inter-task interference for a sporadic task using which, a value which is less pessimistic is computed than compared to computing inter-task interference where the task is modeled as periodic.

8.2.1. Arrival Curves

Firstly, an activation trace is required that simulate the activations of a sporadic task, the algorithm 1 given several activations n, generates an activation trace with n activations, which is an array of absolute times which represent the absolute time at which an instance of the task is activated, assuming that the first instance of the task starts at time = 0. The algorithm generates the trace ensuring the conditions that, as shown in equations Equation 5.7, Equation 5.9 and Equation 5.8, i.e., the minimum distance between two activations is the period, and maximum distance between two activations is two times the period, and for three activations, the minimum distance required for three activations is the sum of the minimum distance of two activations and the period and the maximum distance of three activations is the sum of the distance between two activations and two times the period and so on. The detailed algorithm to generate a random activation trace for a sporadic task is shown in algorithm 1 [21].

Algorithm 1: Generate Trace Uniform

```
  Output: activation_trace
1 def generate_trace_uniform(period: int, number_of_activations: int) ->
   activation_trace: list<float>:
    /* The algorithm simulates an activation trace for a task using uniform
       distribution for randomness                                        */
2   activation_trace = list()        // Activation Trace containing absolute times
3   activation_trace.append(0)   // First activation of the task is at time = 0
4   activation_trace.append(period)     // Second activation at time period, min
     distance between two activations is the period
5   for i in range(2, number_of_activations):
6       activation_trace.append(trace[i - 1] + random.uniform(0, period) + period)
         // The next activation can occur between period and 2*period
7   end
8   return trace
9 end
```

Once an activation trace has been generated, there is a need for methods that reads the entire trace and provide the model that is required to compute the delta and eta functions. Using an external package pyCPA [12], that provide methods to compute a pseudo-conservative

event model from a given activation trace, the line three in the algorithm 2, invokes the pyCPA method that retrieves a TraceEventModel *delta*, an TraceEventModel provide methods *delta_min* and *eta_plus* which are the minimum distance function δ^- and the maximum arrival function η^+, the usage of these methods are done in the DagTask class as shown in Figure 8.1. The functions names are $delta\_minus\_function$ and $eta\_plus\_function$, the implementation is shown in Listing A.1. The TraceEventModel is different for each instance of the class *DagTask*, this model is generated as soon as the task is created and is assigned to the *delta* attribute of the class *DagTask* as shown in Figure 8.1, the methods then uses this *delta* for computing the *delta_min* and *eta_plus* functions.

Algorithm 2: Generate Delta

```
Output: activation_trace, which is an array of absolute times
1 def generate_delta(period) -> delta:
    /* The algorithm generates a model delta using which delta and eta
       functions for a task can be computer                              */
2   activation_trace = generate_trace_uniform(period, 100)          // Generate an
     activation trace of length 100
3   delta = pyCPA.model.TraceEventModel(activation_trace)            // Compute a
     pseudo-conservative event model from a given trace, The algorithm will
     compute delta and eta functions on the trace by evaluating all
     candidates
4   return delta
5 end
```

Finally, these methods are used in the computation of the inter-task interference as shown in Listing A.5. The important difference being, in the Equation 5.5 which uses the period T_i for computation whereas in the Equation 5.6 uses the arrival curve for computation. The $amount\_of\_jobs$, i.e., the number of activations in the interval when the period is used, is calculated using Listing 8.1. When the arrival curve is used, this line can be replaced using Listing 8.2.

Listing 8.1: Amount of jobs computation using Period

```
amount_of_jobs = math.floor(interval + response_time - min_wcet / period) + 1
```

Listing 8.2: Amount of jobs computation using arrival curves

```
amount_of_jobs = eta_plus_function(interval + response_time - min_wcet)
```

8.3. DAG Generators

Random graphs arise when the construction of a graph involves randomness. Random graphs are a suitable choice when there is a need for modeling graphs that are inherently random [49]. To evaluate the performance of the response time calculation algorithm, there is a need for these random Directed Acyclic Graph that allow experimentation with random possible designs of a DAG. This section explains and shows the implementation of two methods used to generate random graphs.

8.3.1. Class Diagram

The class diagram of the implementation of random DAG task builders is shown in the Figure 8.2. The abstract class *DagTaskBuilders* provides variables that are needed by methods required to create a random graph. For instance, *number_of_processors* are used to represent a random number of processors, *minimum_wcet* and *maximum_wcet* are used to create Jobs or subtasks with random WCET in this range. *RenyiDagTaskBuilder* and *ForkJoinDagTaskBuilder* extend the abstract class *DagTaskBuilders*. The detail working of the methods *RenyiDagTaskBuilder* and *ForkJoinDagTaskBuilder* are shown in subsection 8.3.2 and subsection 8.3.3 respectively. If any new *DagTaskBuilders* are to be added, they can be added by extending the *DagTaskBuilders* without having to change any other modules.

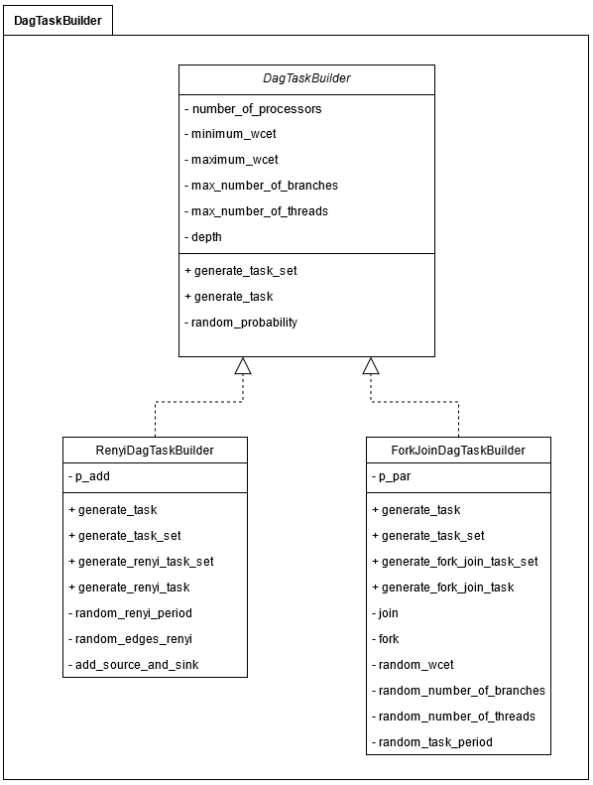


Figure 8.2.: Class Diagram: DagTaskBuilder

8.3.2. The Erdös-Rényi method and The UUniFast method

The method uses the Erdös-Rényi [15] method for generating the graphs, and the UUniFast method [10] is used for utilization computation for the task set.

Erdös-Rényi [15] method defines two methods to generate graphs the $G(n, p)$ and $G(n, M)$ methods [13]. Where the $G(n, p)$ uses the number of vertices n and a probability value, and the $G(n, M)$ method uses the number of vertices n and number of edges M.

Definition 8.3.1 (The $G(n, p)$ method)**.** *For a given n number of nodes, the $G(n, p)$ method generates a random graph where each element of the (n 2) possible edges is present with independent probability p [13].*

The $G(n, p)$ method is adapted for generating DAG task sets [29] [51]. A total utilization value is required to generate the task set. Firstly, a random number of nodes (subtasks) are chosen from a given range, as well as the WCET for each node is also randomly picked from a given range. For each node(subtask) the period is computed using the Equation 8.1 [51] until the required total utilization is exceeded, where U_{tot} represents the total utilization of the task set. Gamma represents the gamma distribution. And the deadline of the task is set to be as same as the calculated Period T_i. For the last node, when the utilization exceeds the total utilization U_{tot}, the Period T_i is adapted such that the task utilization U_i matches the remaining utilization to satisfy the total utilization U_{tot} [51].

$$T_i = \left(L_i + \frac{W_i}{0.4 * U_{tot}} \right) * (1 + 0.25 * Gamma(2, 1)) \tag{8.1}$$

Once the nodes are generated, for each pair of the nodes, an edge is added if a random value within the range (0, 1) is less than a predefined threshold $p\_add = 0.1$. The value of *p_add* determines the maximum level of parallelism of the DAG, i.e., the higher the value of *p_add* higher the probability of the tasks being sequential and having a longer critical path. An example of a DAG task generated using the Erdös-Rényi method is as shown in Figure 8.3.

The Unnifast Algorithm is used for efficiently generating task sets of n tasks with uniform distribution. The algorithm first generates a value of the sum of $n - 1$ variables. Then it sets the first utilization equal to the difference between u_{tot} and the generated utilization. The algorithm keeps generating the random variable "sum of i uniform variables" and computing the single utilization U_i as the difference with the previous sum [10] [9].

Definition 8.3.2 (Total Utilization)**.** *A utilization of a task τ_i is defined as the ratio between the total workload to the period i.e, $U_i = \frac{W_i}{T_i}$. And for a task set τ comprising of n tasks the total utilization can be calculated using the Equation 8.2 [45].*

$$U_{tot} = \sum_{i=1}^{n} U_i \tag{8.2}$$

For generating a task set τ, which contains multiple DAGs the Unnifast Algorithm[9] is used to compute individual task utilization U_i for a fixed number of tasks n.

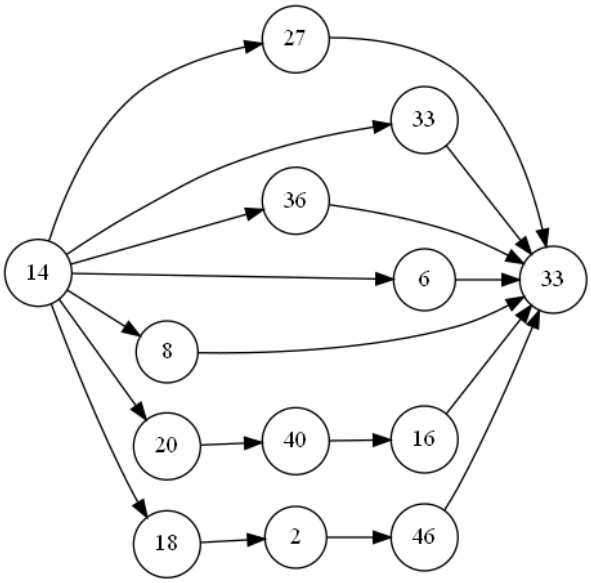


Figure 8.3.: Example DAG generated using The Erdös-Rényi method

To generate a task set τ comprising of n tasks, the following steps are followed [23] for a randomly chosen total utilization U_{tot} in the range of [1, 3] :

1. Firstly, a DAG task is created using the Erdös-Rényi method using the steps given above.
2. Utilization is calculated for each individual task using $U_i = \frac{W_i}{T_i}$
3. The above steps are done until the required total utilization U_{tot} is exceeded
4. Once the required total utilization U_{tot} is exceeded, the following steps are executed to create the last subtask:
   a) Remaining utilization U_{rem} of the subtask is calculated by subtracting the current utilization from the total utilization.
   b) The period of this task is calculated using $T_i = \frac{W_i}{U_{rem}}$ to satisfy the individual task utilization.

The detailed algorithm of generating a task set τ is shown in the algorithm 3 and a few dependent helper methods that are a part of the *DagTaskBuilder* are as shown in the Figure 8.2 are added to the Listing A.6.

Algorithm 3: Generate Renyi Dag Task Set

```
Input: A value randomly selected total_utilization
Output: List of DagTask
def generate_renyi_task_set(total_utilization :  float) -> list<DagTask>:
    /* Generate a task set using Renyi's and Unnifast methods until the
       total utilization is reached                                    */
    current_utilization = 0
     // A variable to keep a track of the utilization of the task set using
     the Unnifast method
    task_set = []
    p_add = 10  // Threshold Probability to determine the number of edges in
     the dag
    while current_utilization <= total_utilization:  /* Generate tasks until
     the required total utilization is exceeded */
        dag_task = generate_renyi_task(total_utilization)     // Generate a dag task
         using renyi method
        if current_utilization + dag_task.get_utilization() <
         total_utilization:     /* Create dag task, and update utilization */
            current_utilization += dag_task.get_utilization()
            task_set.append(dag_task)
        else:     /* Creation of a the last task when utilization exceeds the
         total utilization */
            remaining_utilization = total_utilization - current_utilization        // Use
             remaining utilization to ensure the total utilization is valid
            period = math.ceil(dag_task.workload / remaining_utilization)
            critical_path = dag_task.critical_path
            if period >= critical_path:
                deadline = period
                number_of_threads = math.ceil(dag_task.workload - critical_path /
                 deadline - critical_path)
                if number_of_threads == 0:
                    number_of_threads = 1
                modified_dag_task = DagTask(dag_task.dag, period, deadline,
                 number_of_threads, remaining_utilization)
                task_set.append(modified_dag_task)
                break            // Terminate after creation of the last subtask
    return task_set
end
```

Algorithm 4: Generate Renyi Dag Task

```
Output: A newly created DagTask
def generate_renyi_task(total_utilization: float) -> DagTask:
    /* Generate a task using Renyi's method                              */
    number_of_vertices = random.randint(10, 50) // Random number of subtasks for
     a single task
    jobs = []
    for i in range(number_of_vertices - 2): /* Create random subtasks for a
     single task */
        wcet = random.randint(1, 50)
        job = Job(wcet)
        jobs.append(job)
        dag.add_node(job)                  // Add the created node to the graph
    end
    random_edges_renyi()          // Generate random edges(dependencies) between
     subtasks
    add_source_and_sink()             // Add a single source and sink to the dag
    period = random_renyi_period(workload, critical_path, total_utilization) // Random
     period using renyi and unnifast methods
    utilization = dag.calculate_workload() / period          // Shown in Listing A.4
    deadline = period
    number_of_threads = math.ceil(workload - critical_path / deadline - critical_path)
    return DagTask(dag, period, deadline, number_of_threads, utilization)
end
```

Algorithm 5: Random edges using Renyi

```
def random_edges_renyi(dag: Dag, jobs: list<DagNode>):
    /* Generate random edges for the graph created using Renyi's method  */
    for j1 in jobs:
        for j2 in jobs:
            if j1 == j2:                        /* Restrict self loops in dag */
                break
            if self.random_probability() < self.p_add: /* Generate an edge if
             the random probability is less than a predefined value. Helper
             methods in Listing A.6 */
                dag.add_edge(j1, j2)
        end
    end
end
```

Algorithm 6: Renyi Add Source and Sink

```
  Input: dag: Dag
1 def add_source_and_sink(dag:  Dag):
    /* Add single source and sink to the DAG generated by the Renyi method
       */
2   source = Job(random.randint(1, 50))      // Create a single random source and
     subtask
3   sink = Job(random.randint(1, 50)) dag.add_node(source) // Add source and sink
     to the graph
4   dag.add_node(sink)
5   for node in dag.nodes: /* Add edges from source to the nodes that do not
     have an incoming edge and add edges from nodes which do not have an
     outgoing edge to the sink  */
6     if node.job != source and node.job != sink:
7       if dag.graph.in_degree(node.job) == 0:
8         dag.add_edge(source, node.job)
9       if dag.graph.out_degree(node.job) == 0:
10        dag.add_edge(node.job, sink)
11  end
12 end
```

Algorithm 7: Generate Random Period

```
  Output: Period
1 def random_renyi_period(workload:  int, critical_path:  int,
   total_utilization:  float) -> int:
    /* Generate random period according to Renyi's method                  */
2   first_part = (critical_path + workload) / (0.4 * total_utilization)
3   gamma = np.random.gamma(2, 1)
4   second_part = 1 + (0.25 * gamma)
5   return math.ceil(first_part * second_part)
6 end
```

8.3.3. Nested Fork-Join DAG and The UUniFast method

Similarly, the Erdös-Rényi and the UUniFast method as shown in subsection 8.3.2. The Nested Fork-Join DAG and The UUniFast method uses the Nested Fork-Join DAG [16] method for generating the graphs, and follows the UUniFast method [10] for utilization computation of the task set.

A Nested Fork-Join DAG is a DAG comprised of two nodes connected by a single edge is NFJ. If G_A and G_B are two independent NFJ-DAGs, then the DAG obtained through either of the following operations is also a NFJ-DAG [16] [26]:

1. **Series Merge**: merges the sink of G_A with the source of G_B.
2. **Parallel Merge**: merge the source of G_A with the source of G_B and the sink of G_A with the sink of G_B.

The series merge links two NFJ-DAGs one after another sequentially, i.e., links the sink of one NFJ-DAG to the source of the other NFJ-DAG, whereas the parallel merge juxtaposes two NFJ-DAGs by merging their sources and sinks, i.e., a new node source and sink is created, the newly created source is linked to the two sources of the graph, and similarly the sinks of the two graphs are linked to the newly created sink node. Example of a DAG created using NFJ method is as shown in Figure 8.4

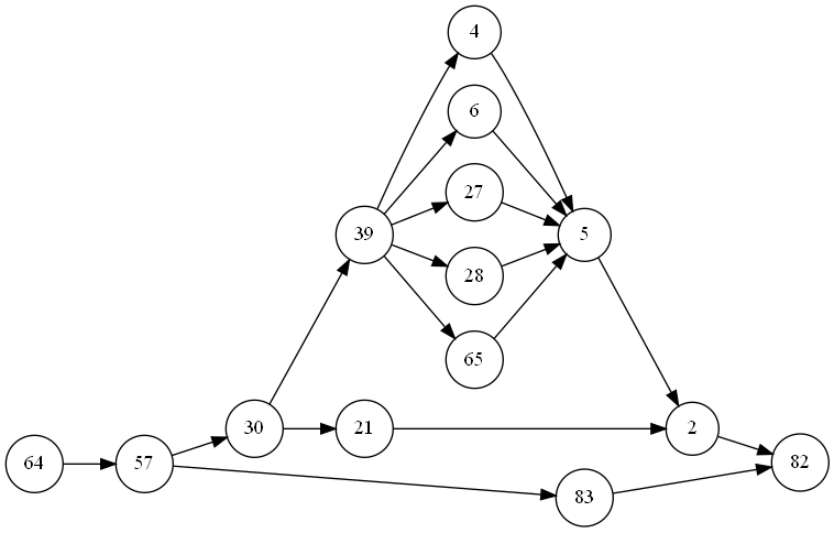


Figure 8.4.: Example DAG generated using Nested Fork-Join method

A node v_i is called a join-node if the number of its ancestors are larger than one i.e, $|ancestors(v_i)| > 1$ and a node v_i is called a fork-node if the number of its descendants are larger than one i.e., $|descendants(v_i)| > 1$. The property that a join-node is a result of parallel merge. Hence, for every join-node v_i there exists a fork-node v_f such that the sub-graph $G^{'}$ has v_f as a source and v_i as a sink, must hold for a NFJ-DAG [16].

Using this property, the following steps are executed to create a single random NFJ-DAG [45] [37]:

1. The algorithm start with an empty *Dag*, and a randomly chosen depth.
2. For each node in a layer, the child node and the edges are generated based on the fork probability p_par, and the number of children are determined by a uniform distribution. This step is carried out recursively from the source, until the random depth reached, which then creates a DAG of the required random depth. This is a sequential merge operation.

3. After a fork operation, join is executed to link the nodes between the layer which leads to parallel executing node, which is a parallel merge operation.

The above algorithm is shown in detail as two method *fork* and *join* as shown in algorithm 8 and algorithm 9 respectively.

Algorithm 8: Fork

```
def fork(dag: Dag, predecessor: Job, depth: int):
    /* Recursive function for creating DAG using sequential merge until
       depth                                                          */
    children = []
    job = Job(self.random_wcet())              // Create a job with random WCET
    dag.add_node(job)
    if predecessor is not None:           /* If there is a predecessor, do a
     sequential merge */
        dag.add_edge(predecessor, job)
    if depth >= 0 and self.random_probability() < self.p_par:    /* Create a
     dag, with random number of descendants */
        number_of_jobs = self.random_number_of_branches()          // Call fork
         recursively for all the jobs and until the depth is reached
        for i in range(number_of_jobs):
            children.append(self.fork(dag, job, depth - 1))
        end
    return self.join(dag, job, children)
end
```

Similarly as explained in subsection 8.3.2, To generate a task set τ comprising of n tasks, the following steps are followed [23] for a randomly chosen total utilization U_{tot} in the range of [1, 3] :

1. Firstly, a DAG task is created using the NFJ-DAG method using the steps given above.
2. Utilization is calculated for each individual task using $U_i = \frac{W_i}{T_i}$
3. The above steps are done until the required total utilization U_{tot} is exceeded
4. Once the required total utilization U_{tot} is exceeded, the following steps are executed to create the last subtask:
   a) Remaining utilization U_{rem} of the subtask is calculated by subtracting the current utilization from the total utilization.
   b) The period of this task is calculated using $T_i = \frac{W_i}{U_{rem}}$ to satisfy the individual task utilization.

Algorithm 9: Join

```
def join(dag: Dag, current: Job, children: list<Job>):
    /* Algorithm for parallel merge of the DAG that was partly generated by
       fork                                                                  */
    if len(children) > 0:
        // If children of the fork node(current) are empty, return the fork
         node(current)
        job = Job(self.random_wcet())          // Create a job with a random WCET
        dag.add_node(job)                  // Add the subtask created to the graph
        for child in children:
            dag.add_edge(child, job)       // Add an edge from all children of the
             fork node(current) to the newly created job creating node in
             parallel
        end
        return job       // Return the newly created job which have no outgoing
         edges
    return current
end
```

The detailed algorithm of generating a task set τ is showed in the algorithm 11.

Algorithm 10: Generate Fork-Join Task

```
  Output: DagTask
def generate_fork_join_task() -> DagTask:
    /* Generate a task using nfj method                                      */
    dag = Dag([])                                        // Create an empty Dag
    j = fork(dag, None, depth) // Invoke forks to create a sequential merge dag
    fork(dag, j, depth)
    workload = dag.calculate_workload()                 // Shown in Listing A.4
    critical_path = dag.get_critical_path()             // Shown in Listing A.2
    period = random_task_period(critical_path, workload) // Shown in Listing A.6
    utilization = workload / period
    min_number_of_threads = math.ceil(workload / period)
    number_of_threads = random_number_of_threads(min_number_of_threads)
     // Shown in Listing A.6
    return DagTask(dag, period, number_of_threads, utilization)
end
```

Algorithm 11: Generate Fork-Join Task Set

```
   Input: total_utilization
   Output: Set of DagTask
1  def generate_fork_join_task_set(total_utilization) -> list<DagTask>:
       /* Generate a task set using nfj and Unnifast methods until the total
          utilization is reached                                              */
2      current_utilization = 0 // A variable to keep a track of the utilization of
        the task set using the Unnifast method
3      task_set = []
4      while current_utilization <= total_utilization:  /* Generate tasks until
        the required total utilization is exceeded */
5          dag_task = self.generate_fork_join_task()      // Generate a dag task using
            Nested Fork-Join method
6          if current_utilization + dag_task.get_utilization() <
            total_utilization:     /* Create dag task, and update utilization */
7              current_utilization += dag_task.get_utilization()
8              task_set.append(dag_task)
9          else:     /* Creation of a the last task when utilization exceeds the
            total utilization */
10             remaining_utilization = total_utilization - current_utilization        // Use
                remaining utilization to ensure the total utilization is valid
11             period = math.ceil(dag_task.workload / remaining_utilization)
12             critical_path = dag_task.critical_path
13             if period >= critical_path:
14                 modified_dag_task = DagTask(dag_task.dag, period,
                    number_of_processors, remaining_utilization)
15                 task_set.append(modified_dag_task)
16                 break            // Terminate after creation of the last subtask
17     return task_set
18 end
```

8.4. Priority Assignment

As shown in chapter 7, three algorithms Topological, Random and WCET priority assignment have been illustrated, this section shows the detailed implementation of the algorithms.

8.4.1. Class Diagram

The class diagram of the Priority Assignment implemented is as shown in Figure 8.5, an interface is created named *PriorityAssignment*, which has just a single method named *assign*, which requires a *DagTask* as an input and returns a *DagTask* wherein the subtasks in the DAG have priorities assigned, all the algorithm must implement the interface, as shown in the Figure 8.5, the algorithms *Random*, *Topological* and *WCET* all implement *PriorityAssignment* and have their corresponding characteristics of assigning priorities to the DAG. Any new algorithms can be added by implementing the interface without needing to change any other modules.

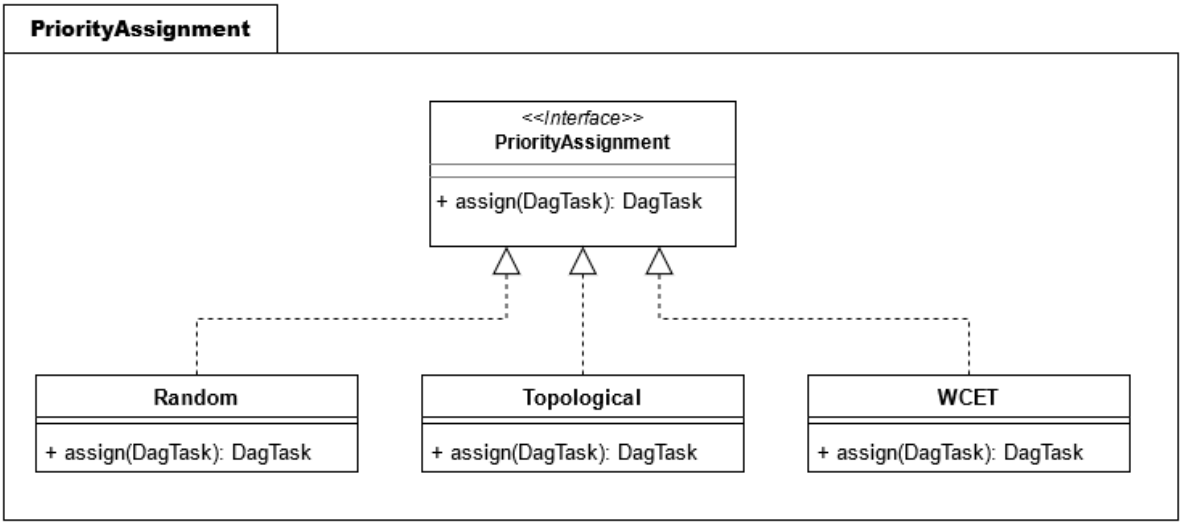


Figure 8.5.: Class Diagram: PriorityAssignment

8.4.2. Topological Priority Assignment

The detailed algorithm for assigning the priorities to a DAG is shown at algorithm 12, the function requires a *DagTask* as an input. It returns the same object of *DagTask* with the corresponding nodes of the DAG of the task having priorities assigned. Line 3 in the algorithm retrieves a list of nodes in topological order of the graph using the NetworkX package [3] and iterates over each node as done in line 5. The priorities are assigned to the nodes sequentially. Once priorities are assigned to the nodes, the graph is updated which is required by the NetworkX package.

Algorithm 12: Topological Priority Assignment

```
Input: task: DagTask
Output: task: DagTask, where the nodes of the DAG have priorities assigned
/* Algorithm for assigning priorities to the nodes of a DAG using
   topological priority assignment method                          */
class Topological:
    def assign(task:  DagTask) -> DagTask:
        nodes = list(nx.topological_sort(task.dag.graph)) // Retrieve the nodes in a
         topological order
        priority_dict = {}
        for i in range(len(nodes):  /* Assign priority to each node and store
         the priorities assigned in a dictionary */
            nodes[i].priority = i
            priority_dict[nodes[i]] = i
        end
        nx.set_node_attributes(task.dag.graph, priority_dict, "priority")      // Update
         graph with priorties with updated nodes
        return task
    end
end
```

8.4.3. Random Priority Assignment

The algorithm 13 requires a *DagTask* wherein priorities are not assigned to the subtasks in the DAG and returns a DagTask with priorities assigned to the subtasks. Firstly the source of the DAG is retrieved using the NetworkX package [3] at line 3 of algorithm 13. The source is added to a list named *parallelism* and also to a set named *task_set*. The nodes in the *parallelism* list corresponds to the nodes that exists in a layer. The first layer is just the source alone. The next steps are carried out until the *task_set* is empty. Line 10 of the algorithm 13 shuffles the list of nodes of a layer randomly, and at the next step the method *assign_priorities_layer* as shown in Listing A.7 is invoked to assign priority sequentially for a layer. After assigning the priorities for a layer the list *parallelism* and the set *task_set* are cleared, the next layer is retrieved and added to both *parallelism* and *task_set*. The entire algorithm is shown at algorithm 13.

Algorithm 13: Random Priority Assignment

```
   Input: task: DagTask
   Output: task: DagTask, where the nodes of the DAG have priorities assigned
1  class Random:
2      def assign(task:  DagTask) -> DagTask:
           /* Algorithm for assigning priorities to the nodes of a DAG using
              random priority assignment method                              */
3          tasks = list(nx.topological_sort(task.dag.graph)) // Retrieve the nodes in a
            topological order
4          source = tasks[0]                                  // Save the source
5          task_set = {source}
6          parallelism = [source]  // parallelism at any time contains the nodes of
            a layer
7          priority = 0
8          priority_dict = {}
9          while len(task_set) != 0:  /* Assign priority to each layer at a time
            and store the priorities assigned of all nodes in a dictionary */
10             random.shuffle(parallelism)  // Shuffle the nodes of a layer randomly
11             assign_priorities_layer(parallelism)   // Assign priorities sequentially
                to the randomly order nodes, method implementation shown at
                Listing A.7
12             get_next_layer(parallelism)          // Method implementation shown at
                Listing A.8
13         nx.set_node_attributes(task.dag.graph, priority_dict, "priority")      // Update
            graph with priorities with updated nodes
14         return task
15     end
16 end
```

8.4.4. WCET Priority Assignment

Similarly, as the method Random Priority Assignment explained in algorithm 13, The algorithm 14 requires a *DagTask* as input wherein priorities are not assigned to the subtasks in the DAG and return a DagTask with priorities assigned to the subtasks. Firstly the source of the DAG is retrieved using the NetworkX package [3] at line 3 of algorithm 14. The source is added to a list named *parallelism* and a set named *task_set*. The nodes in the *parallelism* list correspond to the nodes in a layer. The first layer is just the source alone. The next steps are carried out until the *task_set* is empty. Line 10 of the algorithm 14 sorts the nodes in descending order of a layer based on the WCET of the nodes, and at the next step the method *assign_priorities_layer* as shown in Listing A.7 is invoked to assign priority sequentially for a layer. After assigning the priorities for a layer the list *parallelism* and the set *task_set* are cleared, the next layer is retrieved and added to both *parallelism* and *task_set*. The entire

algorithm is shown at algorithm 13.

The difference between Random Priority Assignment and WCET Priority Assignment is the line 10 of both algorithm 13 and algorithm 14, in algorithm 13 the nodes of a layer are randomly shuffled, whereas in algorithm 14 the nodes in a layer are sorted in descending order based on their WCET.

Algorithm 14: WCET Priority Assignment

```
   Input: task: DagTask
   Output: task: DagTask, where the nodes of the DAG have priorities assigned
1  class WCET:
2      def assign(task:  DagTask) -> DagTask:
           /* Algorithm for assigning priorities to the nodes of a DAG using
              WCET priority assignment method                                  */
3          tasks = list(nx.topological_sort(task.dag.graph)) // Retrieve the nodes in a
            topological order
4          source = tasks[0]
5          task_set = source
6          parallelism = [source]  // parallelism at any time contains the nodes of
            a layer
7          priority = 0
8          priority_dict = {}
9          while len(task_set) != 0:  /* Assign priority to each layer at a time
            and store the priorities assigned of all nodes in a dictionary */
10             parallelism.sort(key=operator.attrgetter('wcet'), reverse=True) // Sorts the
                nodes in a descending order of their WCET
11             assign_priorities_layer(parallelism)   // Assign priorities sequentially
                to the randomly order nodes, method implementation shown at
                Listing A.7
12             get_next_layer(parallelism)          // Method implementation shown at
                Listing A.8
13         nx.set_node_attributes(task.dag.graph, priority_dict, "priority")      // Update
            graph with priorities with updated nodes
14         return task
15     end
16 end
```

8.5. Intra-task interference Bound Calculation

The algorithm for computing the intra-task interference safe upper bound was mentioned in algorithm 17, this sections shows details about the implementation of the algorithm.

8.5.1. Class Diagram

The methods for Computation Algorithms uses the interface *Algorithm*. This interface provides methods to compute inter-task, intra-task interference and response time bounds. Algorithms must implement the interface *Algorithm* to compute the times. Currently, two algorithms are added as shown in Figure 8.6, *SchmidMottok* and *ResearchAlgorithm*. Other algorithms can be added by implementing the interface without having the need to modify any other module logic.

The algorithm uses the following modules for computation:

- Type Function: An interface that contains the methods which contain the equations to compute the inter-task and intra-task interference currently, only a single type function *UnknownStructure* is implemented. Other type functions such as *KnownStructure* can be added based on the system and execution model.

- Priority Manager: For the computation of inter-task interference, tasks of a task set can be preempted and the thread pools of the task are scheduled to the OS based on the Priority Manager algorithm, for instance, RMS or EDF [51]. The class diagram *PriorityManager* as shown in Figure 8.6 contains just a single method *cmp* which allows to find the task with the higher priority based on the type of algorithm.

The algorithm *SchmidMottok* for calculation of inter-task interference uses the period i.e., models the sporadic task as a periodic task and computes the intra-task interference considering the subtasks share the same priority as that of the task.

The algorithm introduced in this thesis is the *ResearchAlgorithm*, which is an extension of *SchmidMottok* with the difference being computation of inter-task interference is done using arrival curves, i.e., does not model a sporadic task as periodic and achieves a less pessimistic inter-task interference bound and the other is the computation of intra-task interference, the *ResearchAlgorithm* assign priority to the subtasks using a *PriorityAssignment* and the intra-task interference time is computed considering the priorities to the subtasks again ensuring a tighter upper bound, methods used by the algorithms are mentioned in the class diagram as shown in Figure 8.6. The computation where priority is considered uses the *Thread* class for simulating the execution of the subtasks of threads.

And the algorithm also uses the following modules for testing and experimentation:

- DagTaskBuilder and DagModel, as explained in the section 8.3 provides a method to create a random task-set. This task-set is used to experiment with computations, i.e., firstly, a random task-set is created and then the algorithm is invoked on this task-set to compute the response times.

- Info is used for the final steps after computing the response times to show if a task-set is schedulable or not.

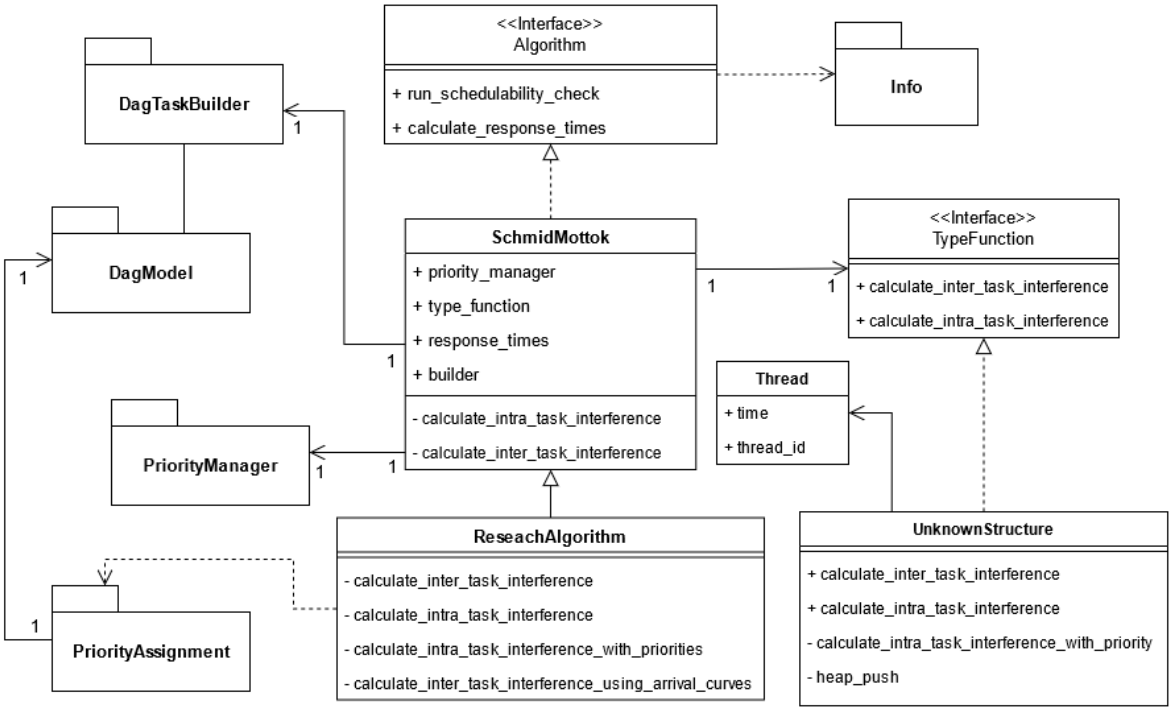


Figure 8.6.: Class Diagram: Algorithm

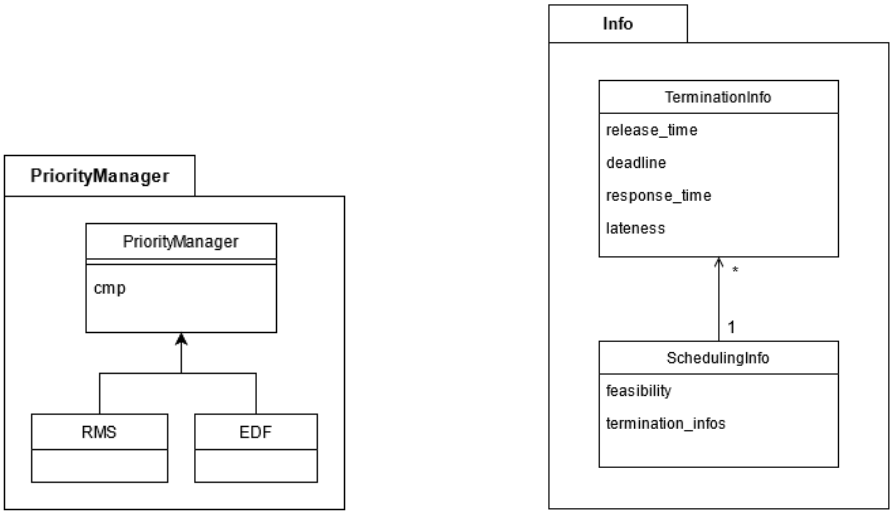


Figure 8.7.: Class Diagram: PriorityManager and Info

8.5.2. Algorithm

The implementation of the algorithm that is used for calculating the upper bound intra-task interference is shown at algorithm 17. The function requires a DAG task with intra-task priorities assigned as input, and the function returns the intra-task interference that occurs for the DAG. The function at algorithm 17 i.e, calculate_intra_task_interference is the main function that is called for computation, the function also uses the helper methods heap_push(algorithm 15) and update_threads(algorithm 16). The function calculate_intra_task_interference works by scheduling the nodes layer by layer. Firstly, the first layer i.e, only the source is scheduled and after that the next layer is scheduled and so on. After collection of a layer, the function uses the helper method heap_push to add the highest priority node in the layer to the most appropriate thread. The threads are implemented using heap queue, where each heap contains the absolute times, simulating the execution of nodes on threads. The detailed explanation and illustration of the working of the algorithm is shown at subsection 6.3.1 and Figure 6.6 respectively.

9. Experiments

This chapter shows the testing and the experiments done to prove the findings of the thesis. Firstly, comparing intra-task interference for random DAGs with and without intra-task priorities shows if assigning priorities has a significant impact. Second, it compares interferences between three types of priority assignment algorithms.

9.1. Subtasks with Priority vs. Subtasks with No Priority

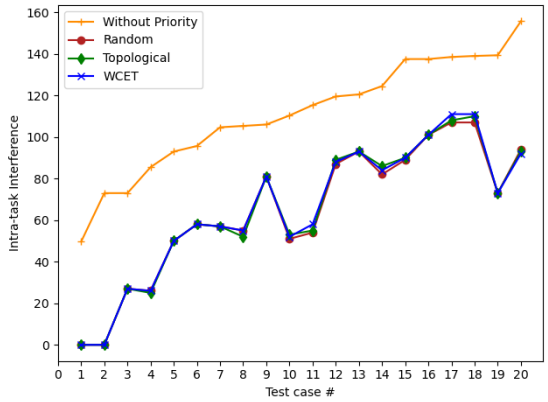
This section illustrates the experiments done to compare random DAGs where subtasks have priorities assigned vs. where subtasks have no priority assigned as shown in chapter 6. The calculation of intra-task interference for subtasks where priorities are assigned follows the algorithm 17 using three priority assignments methods as explained in chapter 7 and calculation of intra-task interference where no priorities are assigned is calculated using the Equation 6.1.

9.1.1. Using The Erdös-Rényi method

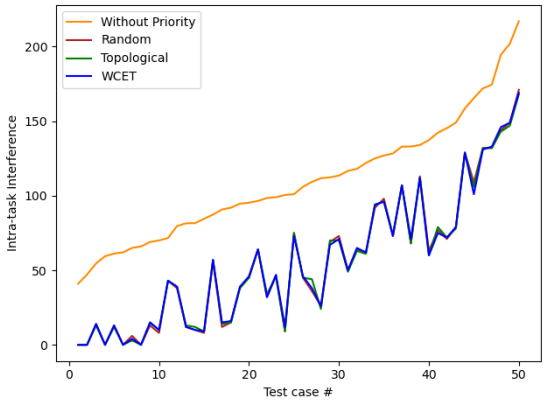
Experiments are performed using random graphs created by The Erdös-Rényi method as shown in algorithm 4 with the following constraints:

- Single source and sink DAG
- p_add = 10, which is used for random edges in the Erdös-Rényi method
- total_utilization = 8, which restricts the number of threads to 8
- WCET in the interval range(1, 50)

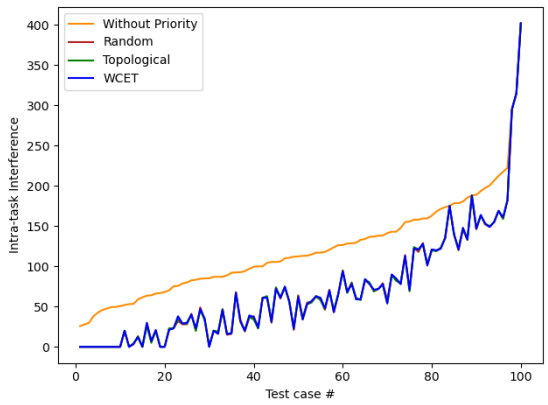
Experiment 9.1.1. The experiment is done using random DAG tasks created using The Erdös-Rényi method as shown in subsection 8.3.2 with the restrictions mentioned above, the charts shown in Figure 9.1.1 compares the intra-task interference where one line "Without Priority" represents the calculation of intra-task interference using Equation 6.1 and the other three "Random, Topological, and WCET" are computed using algorithm 17. The x-axis represents the Test case number, and the y-axis represents the intra-task interference. Each test case is different, where the number of nodes is picked at random in the interval range(50, 100), similarly the number of threads in the interval range(1, 8), and WCET in the range of (1, 50). The results of the calculations are sorted using the computations of "without priority" as a reference.



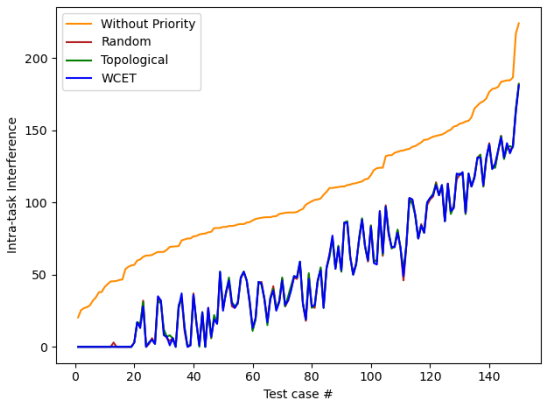
(i) Number of tests cases = 20



(ii) Numberof tests cases = 50



(iii) Number of tests cases = 100

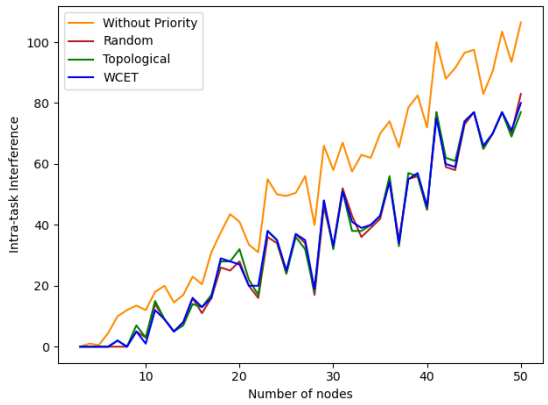


(iv) Number of tests cases = 150

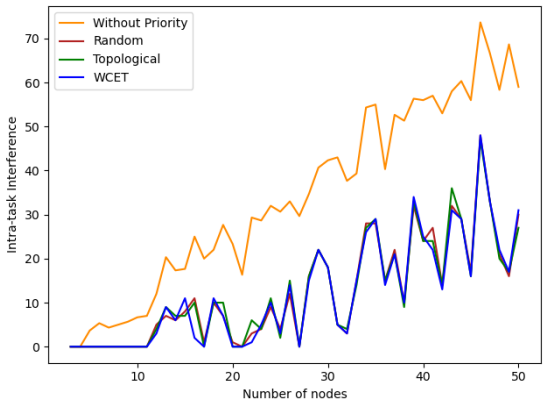
Figure 9.1.: With vs. Without Priority for random DAG generated using Erdös-Rényi method

Result: As shown in the Figure 9.1.1 there are four charts with a fixed number of test cases, where chart (i) contains 20 random tests DAG and similarly (iv) includes 150 random tests. These tests are executed to cover all types of design and configuration of DAG that can be created using Erdös-Rényi method. The impact of the charts illustrated in shows that the safe upper bounds of the intra-task interference can be tightened by assigning priorities, i.e., with intra-task priorities. There are instances in the charts where intra-interference is the same for all methods; this happens in a scenario where the number of threads is one.

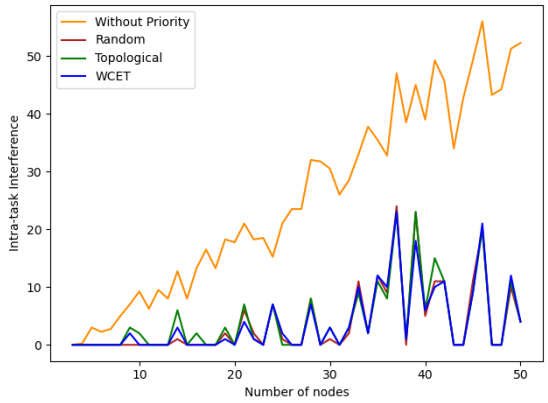
Experiment 9.1.2. As shown in Experiment 9.1.1, the safe upper bound of the intra-task interference can be tightened by assigning priorities. This experiment compares the same computation methods as the Experiment 9.1.1 but shows the effects on the intra-task interference when the number of threads is kept constant for a variable number of nodes. For the chart illustrated in Figure 9.2, the x-axis represents a test case and the number of nodes. For instance, 10 on the x-axis means the 10th random test case with a DAG randomly generated using 10 nodes. The y-axis represents the intra-task interference. The experiments are executed keeping a fixed number of threads. And the maximum possible number of nodes is 50, i.e., the range of the x-axis begins with a DAG with 3 nodes to a DAG with 50 nodes.



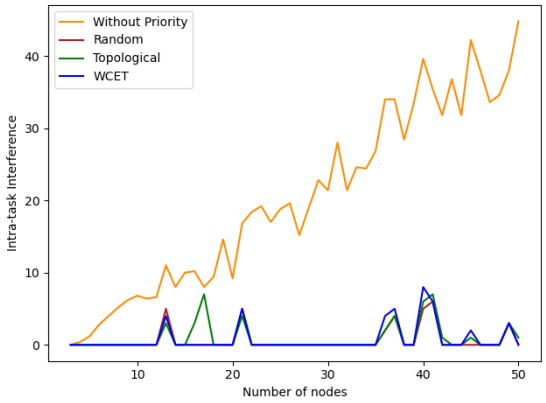
(i) Number of threads = 2



(ii) Number of threads = 3



(iii) Number of threads = 4

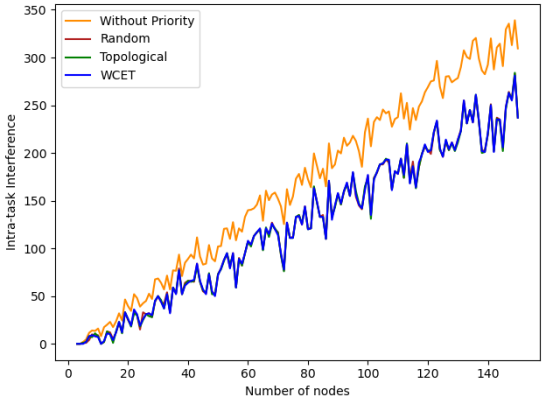


(iv) Number of threads = 5

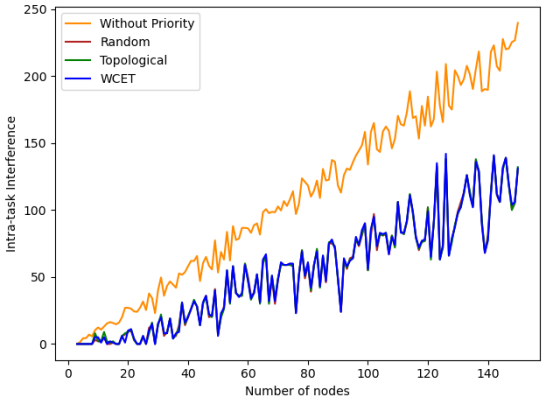
Figure 9.2.: With vs. Without Priority for random DAG generated using Erdös-Rényi method using a fixed number of threads and 50 nodes

Result: This experiment's result again shows that the upper bound intra-task interference can be tightened by assigning priorities. In addition, as the number of threads increases, the charts illustrate that the average difference between the results of the computation methods also increases. Again proving that assigning intra-task priorities allows for a lower response time.

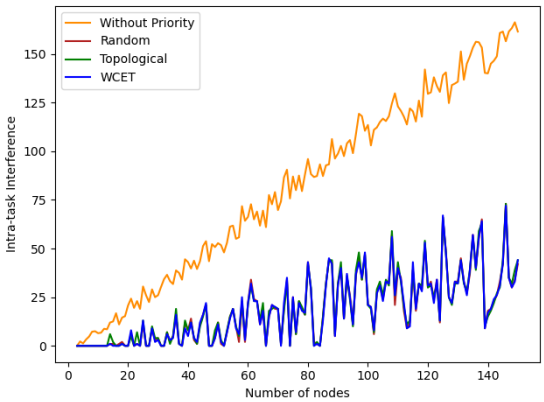
Experiment 9.1.3. Extending the Experiment 9.1.2, with an increment in the number of nodes from 50 to 150. Again, the number on the x-axis represents a test case and the number of nodes. For instance, the value 120 in the x-axis of the graphs represents a DAG with 120 nodes and the 120th random test case.



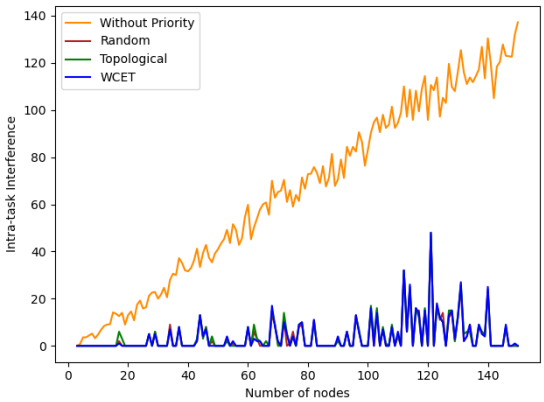
(i) Number of threads = 2



(ii) Number of threads = 3



(iii) Number of threads = 4

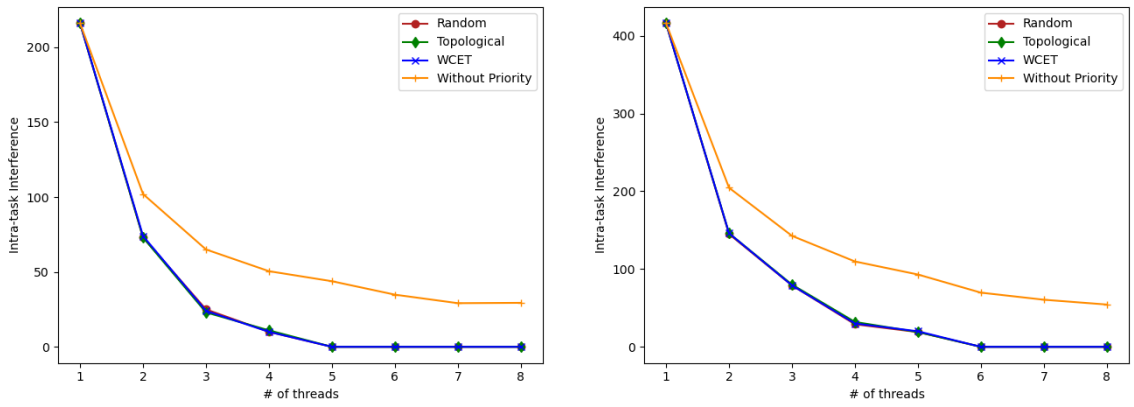


(iv) Number of threads = 5

Figure 9.3.: With vs. Without Priority for random DAG generated using Erdös-Rényi method using a fixed number of threads and 120 nodes

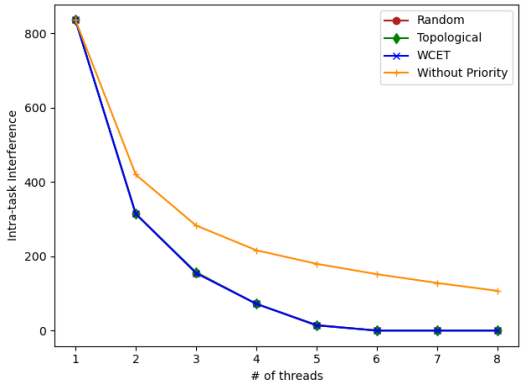
Result: The result of this experiment again shows that the upper bound of the intra-task interference can be tightened by assigning priorities. And also, as the number of threads increases, the average difference between the intra-interference with priority vs. without priority increases as same as the results of Experiment 9.1.2.

Experiment 9.1.4. As shown in Experiment 9.1.1, the upper bound of the intra-task interference can be tightened by assigning priorities. This experiment shows the effects of increasing the threads for the same number of nodes. The charts shown in Figure 9.4 compares the intra-task interference where one line "Without Priority" represents the calculation of intra-task interference using Equation 6.1 and the other three "Random, Topological, and WCET" are computed using algorithm 17. The x-axis in the Figure 9.4 represents the number of threads, and the y-axis represents the computation of the intra-task interference. There are eight test cases where each number on the x-axis represents the test case and the number of threads. The number five on the x-axis represents a random DAG created using Erdös-Rényi method for a fixed number of nodes when scheduled using five threads.



(i) Number of nodes = 50

(ii) Number of nodes = 100



(iii) Number of nodes = 200

Figure 9.4.: With vs. Without Priority for random DAG generated using Erdös-Rényi method with a fixed number of nodes

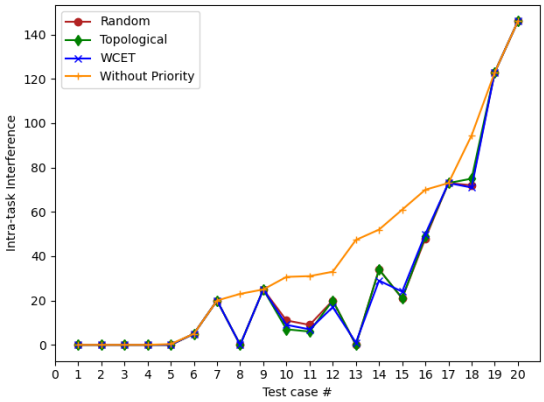
Result: This experiment again confirms that using intra-task priorities allows us to tighten the intra-task interference.

9.1.2. Nested Fork-Join

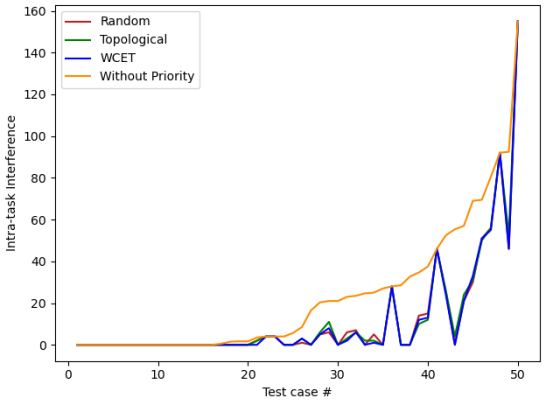
The section now uses the Nested Fork-Join method to conduct experiments, using the same comparisons as shown in subsection 9.1.1. Experiments are performed using random graphs created by Nested Fork-Join method as shown in algorithm 10 with the following constraints:

- Single source and sink DAG
- p_par = 50, which is used for creating random number branches with the range of possible branches in the interval (2, 5)
- Number of threads in the interval range(1, 8)

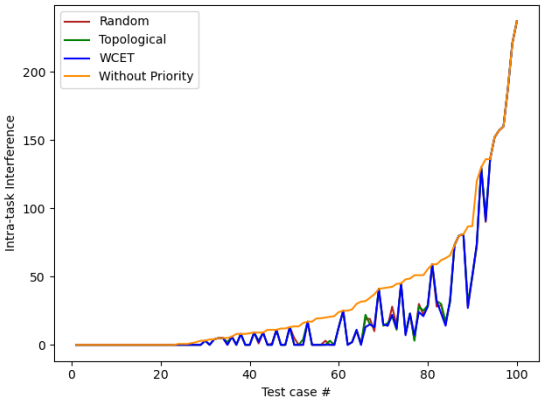
Experiment 9.1.5. The experiment is done using random DAG tasks created using the Nested Fork-Join method as shown in subsection 8.3.3 with the restrictions mentioned above, the charts shown in Figure 9.5 compares the intra-task interference where one line "Without Priority" represents the calculation of intra-task interference using Equation 6.1 and the other three "Random, Topological, and WCET" are computed using algorithm 17. The x-axis represents the Test case number, and the y-axis represents the intra-task interference. Each test case is different, where the depth of the DAG required by NFJ method is set to 5. The results of the calculations are sorted using the computations of "without priority" as a reference.



(i) Number of test cases = 20



(ii) Number of test cases = 50

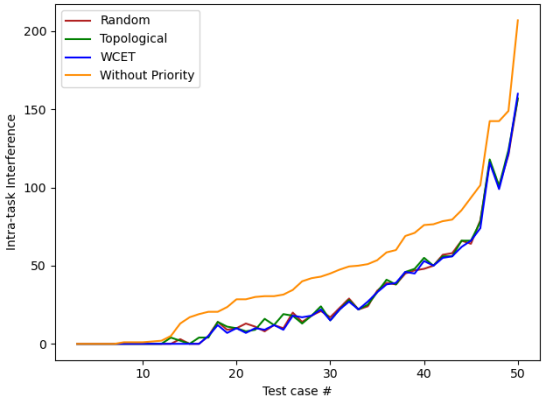


(iii) Number of test cases = 100

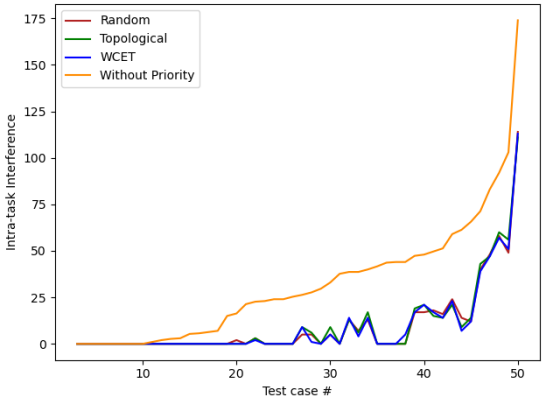
Figure 9.5.: With vs. Without Priority for random DAG generated using Nested Fork-Join method

Result: As shown in the Figure 9.5 there are three charts with a fixed number of test cases, where chart (i) contains 20 random tests DAG and similarly (iii) includes 100 random tests. These tests are executed to cover all types of design and configuration of DAG that can be created using Nested Fork-Join method. The charts illustrated in Figure 9.5 show that the safe upper bounds of the intra-task interference can be tightened by assigning priorities, i.e., with intra-task priorities. There are instances in the charts where intra-interference is the same for all methods; this happens in a scenario where the number of threads is one. The results of this experiment prove the same result as that of the Experiment 9.1.1.

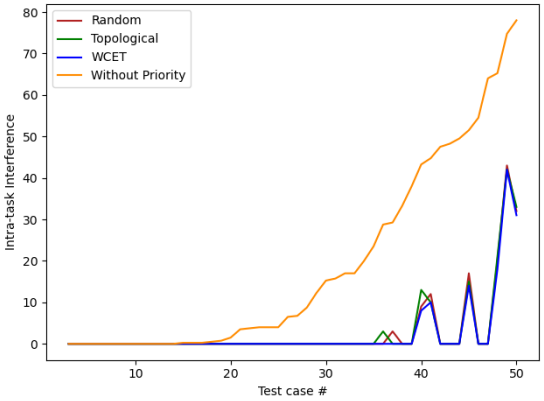
Experiment 9.1.6. As shown in Experiment 9.1.5, the safe upper bound of the intra-task interference can be tightened by assigning priorities. This experiment compares the same computation methods as the Experiment 9.1.5 but shows the effects on the intra-task interference when the number of threads and the depth are kept constant. For the charts illustrated in Figure 9.6, the x-axis represents a test case. The y-axis represents the intra-task interference. The experiments are executed keeping a fixed number of threads. Each test case is different, where the depth of the DAG required by NFJ method is set to 3. The results of the calculations are sorted using the computations of "without priority" as a reference.



(i) Number of threads = 2



(ii) Number of threads = 3

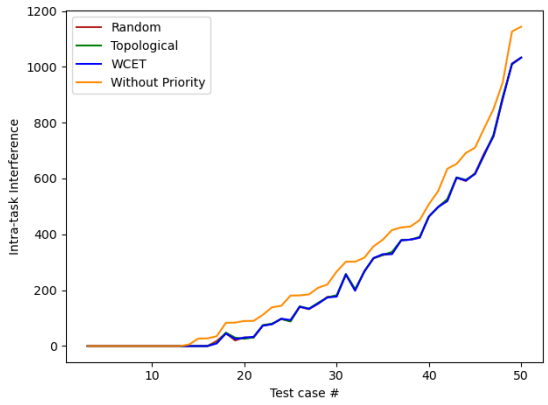


(iii) Number of threads = 4

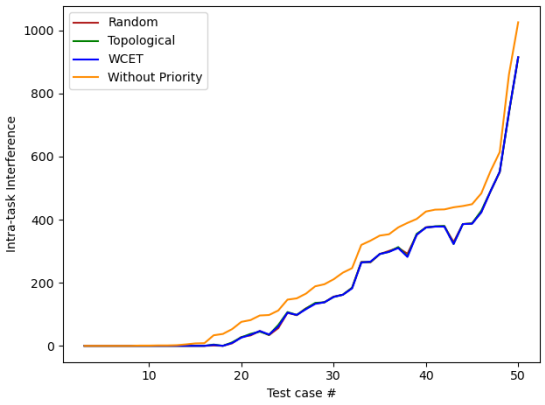
Figure 9.6.: With vs. Without Priority for random DAG generated usingNested Fork-Join method with a fixed number of threads and depth 3

Result: This experiment's result again shows that the upper bound intra-task interference can be tightened by assigning priorities. In addition, as the number of threads increases, the charts illustrate that the average difference between the results of the computation methods also increases. Again proving that assigning intra-task priorities ensures a lower response time.

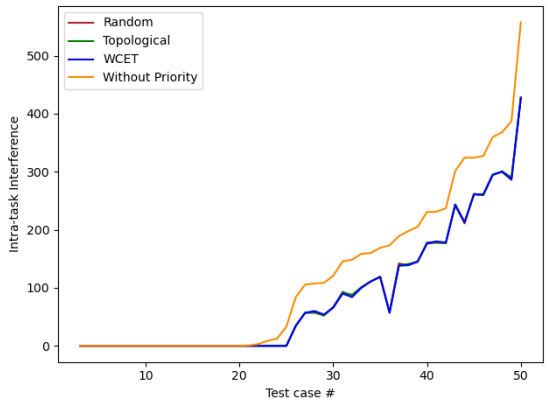
Experiment 9.1.7. Extending the Experiment 9.1.6 with an increment in the depth from 3 to 5.



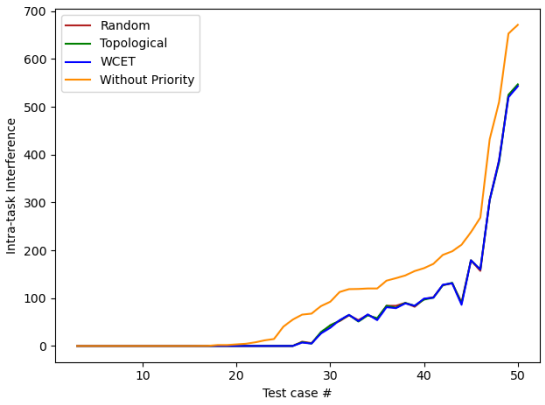
(i) Number of threads = 3



(ii) Number of threads = 4



(i) Number of threads = 5



(ii) Number of threads = 8

Figure 9.7.: With vs. Without Priority for random DAG generated usingNested Fork-Join method with a fixed number of threads and depth 5

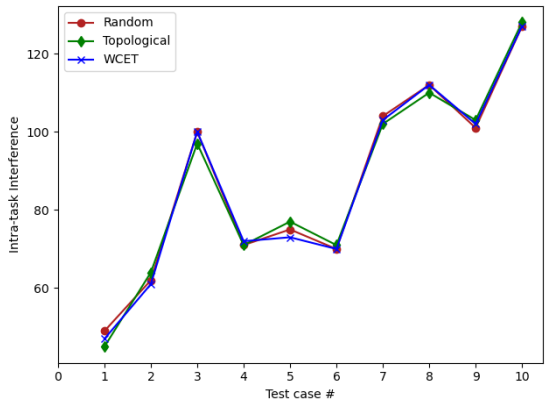
Result: This experiment again shows that the upper bound of the intra-task interference can be tightened by assigning priorities as same as the results of the Experiment 9.1.6.

9.2. Comparing Priority Assignments

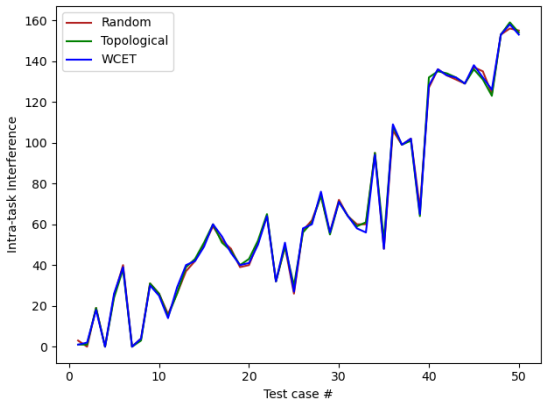
This section contains experiments that specifically demonstrate the difference between the intra-task priority assignments. The previous experiments shown at section 9.1 mainly pinpoints the difference between the computations with and without intra-task priorities. However, the charts in the experiments do not show a clear-cut difference between the three priority assignment algorithms. As shown in the charts at section 9.1, the real difference between Random, Topological, and WCET cannot be figured out.

9.2.1. Using The Erdös-Rényi method

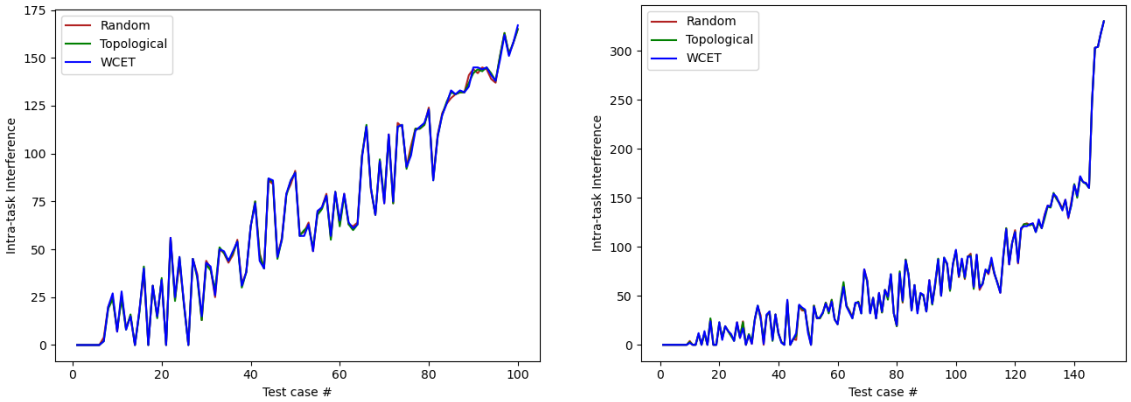
Experiment 9.2.1. This experiment is done to understand if there is a priority assignment that constantly achieves a tighter upper bound intra-task interference compared to others. As shown in Experiment 9.1.1, the upper bound of the intra-task interference can be tightened by assigning priorities. This experiment only compares the intra-task interference where subtasks have priority using three methods mentioned in chapter 7. The x-axis represents the Test case number, and the y-axis represents the intra-task interference. Each test case is different, with a random number of nodes in the range of (50, 100), a random number of threads in the range of (1, 8), and WCET in the range of (1, 50). The results of the calculations are sorted, keeping the result of "without priority" as a reference but not displayed in the graph.



(i) Number of test cases = 10



(ii) Number of test cases = 50

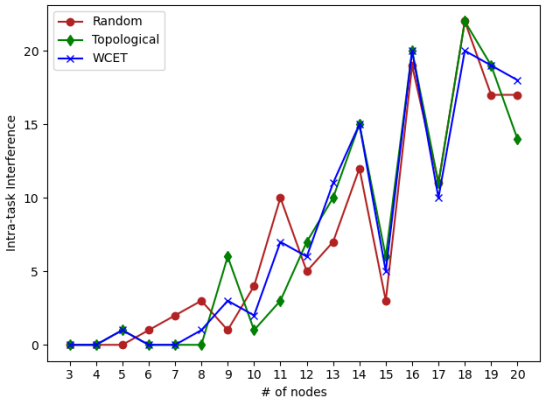


(iii) Number of test cases = 100 (iv) Number of test cases = 150

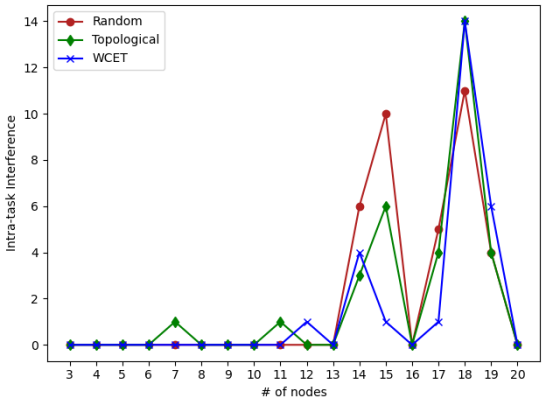
Figure 9.8.: Compare Priority assignments for random DAG generated using Erdös-Rényi method

Result: The experiment results show that no priority assignment algorithm can guarantee a shorter intra-task interference compared to other priority assignment algorithms. These graphs also prove that the average difference between each priority assignment is +-5 time units. In general, there are specific designs of DAG for which an algorithm gives a better result. However, this is not in this thesis's scope and represents future work to find which algorithm can always result in shorted intra-task interference.

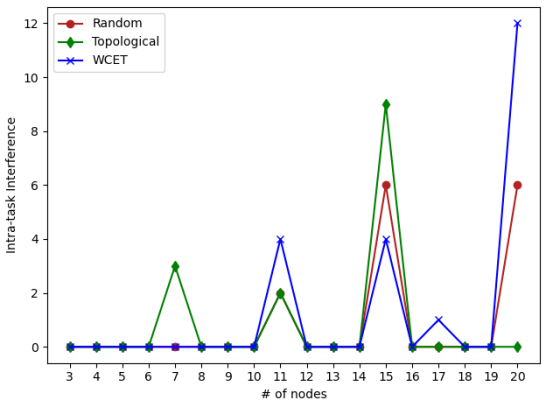
Experiment 9.2.2. This experiment only compares the intra-task interference where subtasks have priority using three methods mentioned in chapter 7. As shown in Experiment 9.1.2, the difference between the upper bounds of with priority vs. without priority intra-task interference increases as the number of threads increases. For the graphs shown in Figure 9.9, the x-axis represents a test case and the number of nodes. For instance, the number 10 on the x-axis represents the 10th random test case with a DAG generated using 10 nodes. The y-axis represents the intra-task interference. The experiments are executed keeping a fixed number of threads. And the maximum possible number of nodes is 50, i.e., the axis range is from a DAG with 3 nodes to a DAG with 50 nodes.



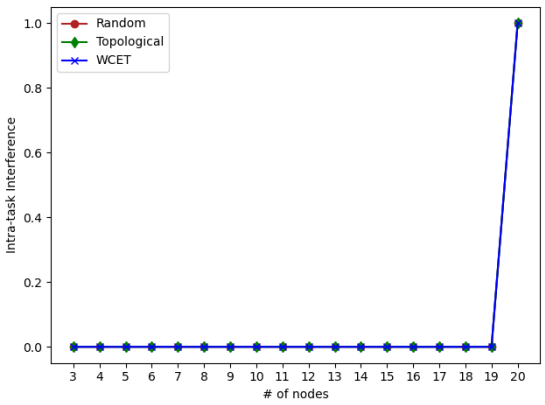
(i) Number of threads = 2



(ii) Number of threads = 3



(iii) Number of threads = 4

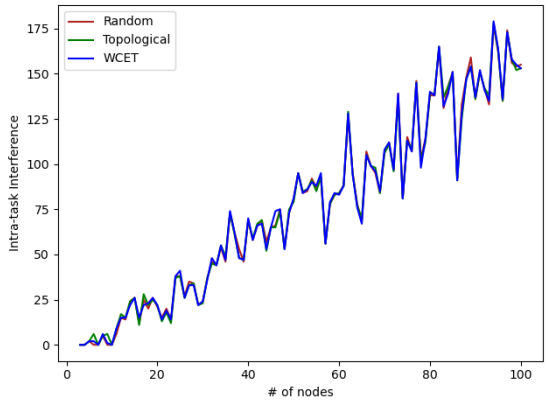


(iv) Number of threads = 5

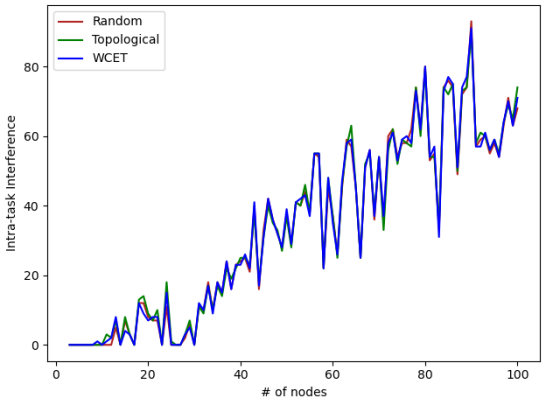
Figure 9.9.: Compare Priority assignments for random DAG generated using Erdös-Rényi method with fixed number of threads

Result: The experiment's results show again that no priority assignment algorithm can guarantee a shorter intra-task interference compared to other priority assignment algorithms. These graphs also prove that the average difference between each priority assignment is +-5 time units. There is no pattern found as the number of threads increases.

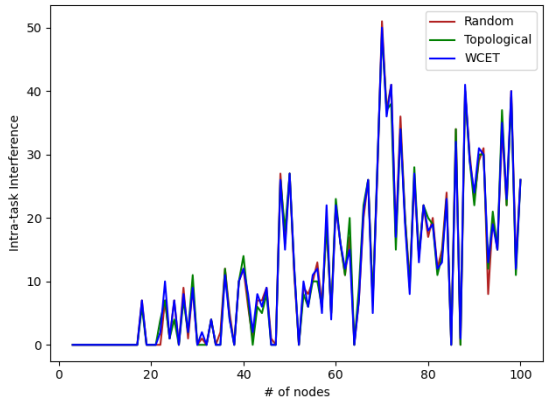
Experiment 9.2.3. Extending the Experiment 9.2.2, this experiment shows by incrementing the nodes from 50 to 100. Again, the number on the x-axis represents a test case and the number of nodes. For instance, the value 80 in the x-axis of the graphs represents a DAG with 80 nodes and the 80th random test case.



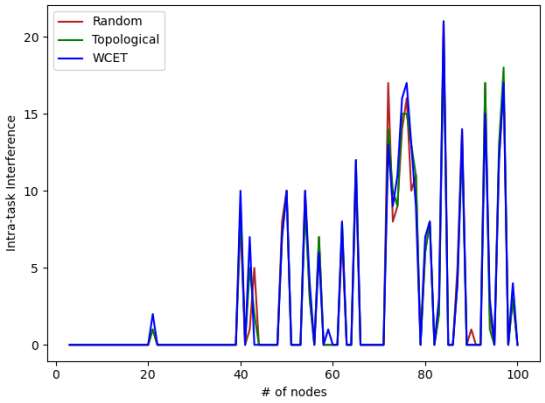
(i) Number of threads = 2



(ii) Number of threads = 3



(iii) Number of threads = 4

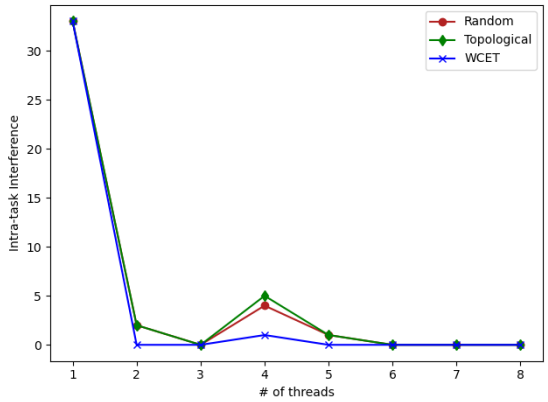


(iv) Number of threads = 5

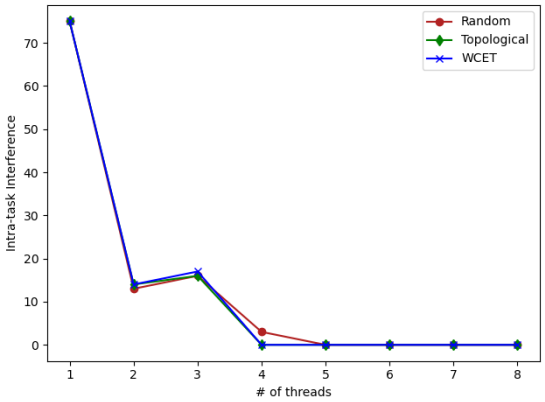
Figure 9.10.: Compare Priority assignments for random DAG generated using Erdös-Rényi method with fixed number of threads

Result: The results of the graphs is as same as the results of the Experiment 9.2.2.

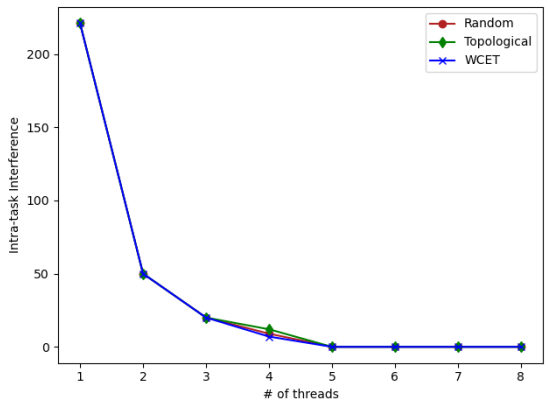
Experiment 9.2.4. This experiment shows the effects of increasing the threads for the same number of nodes. The charts shown in Figure 9.11 compares the intra-task interference of three priority assignments namely "Random, Topological, and WCET" that are computed using algorithm 17. The x-axis in the Figure 9.11 represents the number of threads, and the y-axis represents the intra-task interference computation. There are eight test cases where each number on the x-axis represents the test case and the number of threads. The number five on the x-axis represents a random DAG created using NFJ method for a fixed number of nodes when scheduled using five threads.



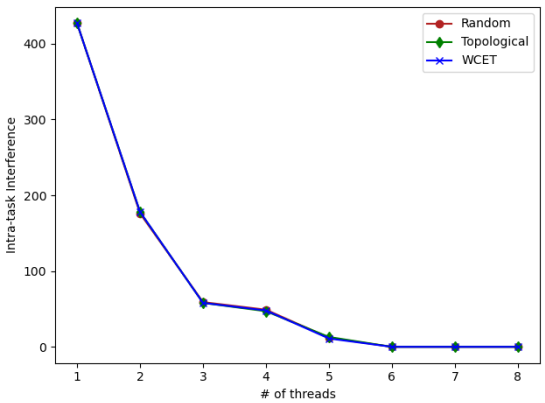
(i) Number of nodes = 10



(ii) Number of nodes = 20



(iii) Number of nodes = 50



(iii) Number of nodes = 100

Figure 9.11.: Compare Priority assignments for random DAG generated using Erdös-Rényi method with fixed number of nodes

Result: The experiment results show that no priority assignment algorithm can always guarantee a shorter intra-task interference compared to other priority assignment algorithms. There is no visible pattern that allows one to confirm if an algorithm is better compared to the others or not.

10. Conclusion

The proposed model allows the scheduling of a task set comprising of sporadic DAG tasks using thread pools on a preemptive global fixed priority system, wherein the subtasks of the DAG task with intra-task priorities are scheduled to the worker threads of the thread pool in a non-preemptive manner. The study proves that a tighter response time bound can be achieved for a DAG task model by assigning intra-task priorities, i.e., priorities to the subtasks or nodes of the DAG. This bound is derived using an algorithm that computes a safe upper bound worst-case intra-task interference that uses the limited task parallelism provided by the thread pools. Along with the arrival curves, the inter-task interference can be computed less pessimistically by not modeling a sporadic task to be periodic. These are confirmed by experiments conducted using random DAG generators that illustrate that the proposed analysis outperforms the state-of-the-art methods.

11. Future Work

Modern real-time systems contain tasks wherein, for instance, a subtask is dependent on multiple inputs. There are cases when the subtask can execute if any of the inputs are ready, i.e., the system model of the Directed Acyclic Graph (DAG) must allow for the OR semantic. In other words, investigating the task interference when a subtask or a node can execute if any one of its ancestors has completed its execution is a future work that can be examined to cover more real-world scenarios of real-time systems as the model mentioned in this work is used for DAG with AND semantics.

In addition, the system model introduced in this work, a DAG with multiple sources and having the same period, in such a scenario, a dummy source is added; however, in the case of multiple sources with different periods, a dummy source cannot be just added, which is, therefore, another improvement that can be investigated in the system model.

The intra-task priority assignment algorithms illustrated in this thesis restrict nodes' priority order such that the priorities do not conflict with the topological order of the Directed Acyclic Graph (DAG). Some studies provide improved priority assignment algorithms where a few algorithms remove this restriction which is mentioned in state of the art; the studies of the literature provide the investigation of a safe upper bound but for a different system model, applying these algorithms to the model presented in this thesis is another improvement that can be made to tighten the upper bound of the worst case response time even further.

An analysis of an execution model in which the subtasks of a DAG are grouped based on priority, and for each priority group, a set of threads are assigned in which the subtasks can be dispatched and executed, which represents a new execution model that can be investigated to check if the response time can be made shorter.

The model presented uses global scheduler for the task-set. An analysis can be made to check if partitioned scheduler can improve the response times.

In recent years, studies have been increasingly conducted for the computation of the response time for scheduling a DAG task on a multi-core processor using different system models, different execution models, and with or without priority assignments; using these studies, a timing analysis toolchain can be developed which combines the result of all literature making it easier for researchers to understand the work done in the years.

A. Appendix: Helper Methods Implementation

Listing A.1: DagTask Class

```
class DagTask(Task):
    def __init__(self, dag: Dag, period: int, number_of_threads: int,
        ↪ utilization, **kwargs):
        deadline = kwargs.get('deadline') if 'deadline' in kwargs else period
        workload_distribution = kwargs.get('workload_distribution') \
            if 'workload_distribution' in kwargs else dag.
                ↪ calculate_workload_distribution()
        super().__init__(period, deadline, dag.calculate_workload(), dag.
            ↪ get_critical_path(), number_of_threads)
        self.dag = dag
        self.workload_distribution = workload_distribution
        self.utilization = utilization
        self.delta = self.dag.generate_delta(period)

    def get_maximum_parallelism(self):
        max_parallelism = 0
        for segment in self.workload_distribution:
            if max_parallelism < segment.height:
                max_parallelism = segment.height
        return max_parallelism

    def delta_minus_function(self, number_of_activations):
        return self.delta.delta_min(number_of_activations)

    def eta_plus_function(self, time_interval):
        return self.delta.eta_plus(time_interval)
```

Listing A.2: Compute Critical Path

```
def get_critical_path(self):
    """ Function to compute the critical path of the given dag, works only for
        ↪ a dag with single source and sink
    """
    critical_path = 0
    nodes = list(nx.topological_sort(self.graph))
    for node in nodes:
        in_edges = self.graph.in_edges(nbunch=node)
        longest_relative_completion_time = 0
        for (source, target) in in_edges:

            if longest_relative_completion_time < source.
                ↪ relative_completion_time:
                longest_relative_completion_time = source.
                    ↪ relative_completion_time

        node.relative_completion_time = longest_relative_completion_time + node.
            ↪ wcet
        critical_path = node.relative_completion_time

    return critical_path
```

Listing A.3: Compute Workload Distribution

```
def calculate_workload_distribution(self):
    """ Function to retrieve segments of workload distribution the given dag
    """
    critical_path = self.get_critical_path()
    segments = []
    segment_duration = 0
    segment_height = 1
    for t in range(critical_path):
        current_height = 0
        for node in self.nodes:
            job = node.job
            if job.relative_completion_time - job.wcet <= t < job.
                ↪ relative_completion_time:
                current_height += 1
        if current_height == segment_height:
            segment_duration += 1
        else:
            segments.append(Segment(segment_duration, segment_height))
```

```
            segment_duration = 1
            segment_height = current_height
    segments.append(Segment(segment_duration, segment_height))
    return segments
```

Listing A.4: Compute Workload

```
def calculate_workload(self):
    workload = 0
    for node in self.nodes:
        workload += node.job.wcet

    return workload
```

Listing A.5: Inter-task interference computation

```
def calculate_inter_task_interference(self, task: DagTask, response_time,
    ↪ interval, parallelism,
                                      use_arrival_curves=False):
        min_wcet = task.critical_path
        period = task.period

        amount_of_jobs = math.floor(interval + response_time - min_wcet /
            ↪ period) + 1

        if use_arrival_curves:
            amount_of_jobs = task.eta_plus_function(interval + response_time -
                ↪ min_wcet)

        workload = 0

        for segment in task.workload_distribution:
            number_of_threads = task.number_of_threads
            if number_of_threads >= parallelism:
                workload += (segment.height * segment.width) / number_of_threads

        interference = amount_of_jobs * workload
        return interference
```

Listing A.6: Snippets of DagTaskBuilder

```
class DagTaskBuilder:
    number_of_processors = 8
    minimum_wcet = 1
    maximum_wcet = 100
    max_number_of_branches = 5
    max_number_of_threads = number_of_processors
    depth = 2
    p_par = 40
    p_add = 10

    def random_probability(self):
        # Default distribution for randInt is Uniform Distribution
        return random.randint(0, 100)

    def get_beta(self):
        return 0.035 * self.number_of_processors

    def random_number_of_branches(self):
        return random.randint(2, self.max_number_of_branches)

    def random_wcet(self):
        return random.randint(self.minimum_wcet, self.maximum_wcet)

    def random_task_period(self, critical_path: int, workload: int):
        return random.randint(critical_path, int(workload/self.get_beta()))

    def random_number_of_threads(self, min_number_of_threads):
        return random.randint(min_number_of_threads, self.max_number_of_threads)
            ↪
```

Listing A.7: Assign Priorities for a given Layer

```
def assign_priorities_layer(parallelism):
    """ parallelism at any time contains the nodes of a layer, and assign
        ↪ priorities to the nodes of a layer
    """
    for i in range(len(parallelism)):
        # Assign priorities sequentially node by node
        assign_priority_job = parallelism[i]
        assign_priority_job.priority = priority
        priority_dict[assign_priority_job] = priority
        priority += 1
```

Listing A.8: Retrieve next layer

```
def get_next_layer(parallelism):
    """ Creates the next layer of the dag and is appended to parallelism and
        ↪ task_set
    """
    parallelism.clear()
    children = []
    # Extract all the descendants from the current layer, available at task_set
    for node in task_set:
        # Set visited of the current layer nodes as True
        node.visited = True
        children.extend(list(nx.neighbors(task.dag.graph, node)))
    task_set.clear()
    # For each child, check if the child node is ready or not, and add to the
        ↪ parallelism and task_set
    for child in children:
        # Check if all the ancestor of the child are visited
        ancestors = list(task.dag.graph.predecessors(child))
        add_child = True
        for ancestor in ancestors:
            if not ancestor.visited:
                # If a child is not ready
                add_child = False
                break
        if add_child:
            # Add children to the new layer if they are ready
            if child not in parallelism:
                parallelism.append(child)
                task_set.add(child)
```

B. Appendix: Intra-task interference Upper Bound Calculation Implementation

Algorithm 15: Heap Push

```
  Input: task: DagTask, node, threads
1 def heap_push(task:  DagTask, node, threads):
    /* Insert the given node to one of the threads, i.e., execute the node
       on a thread                                                          */
2   ancestors = predecessors(node)      // Check if the node can be added to the
     threads
3   add_node = True
4   max_accumulate = -1
5   max_accumulate_node = None
6   for ancestor in ancestors: Comment*[f]Add node to threads only if all its
     ancestors have finished their execution
7     if not ancestor.visited:
8       add_node = False
9       break
10  end
11  if add_node:           // If the node is ready to be added to the threads
12    node.visited = True          // Indicate that the node has been visited
13    for ancestor in ancestors:    // Find the ancestor that completed its
       execution the latest, which indicates that the child can start
       running only after all its ancestor have completed their execution
       or check the ancestor that was executed last i.e., having the max
       accumulate
14      if ancestor.accumulate >= max_accumulate:
15        max_accumulate_node = ancestor
16    end
```

```
17
18      if add_node:
19          if max_accumulate_node is None:  // If a node has no ancestor, can be
             added to the thread with min time
20              thread = heapq.heappop(threads)   // Heapq pop gives the thread with
                 the least time
21              thread.time += node.wcet
22              heapq.heappush(threads, thread) // Using heapq, push the node to the
                 popped thread
23              node.accumulate = thread.time
                 // Accumulate is the thread time at which the node completes its
                 execution
24              node.thread_id = thread.thread_id
25          else:           // If a node has ancestors, i.e., it has dependencies
26              min_value = 99999
27              min_thread_id = -1
28              cond = None
29              for thread in threads:   // Select the thread that the node can be
                 added to
30                  if thread.time >= max_accumulate_node.accumulate:    // This is
                     the case which ensures that a thread time has exceeded the
                     max accumulate so the node can be added to this thread
31                      if min_value > thread.time:  // Choose a thread from the set
                         of accepted threads which has min time, work conserving
32                          min_value = thread.time
33                          min_thread_id = thread.thread_id
34                  if thread.time < max_accumulate_node.accumulate: // This is the
                     case for the threads where the time is still less than the
                     max accumulate
35                      if thread.time + max_accumulate_node.wcet <=
                         max_accumulate_node.accumulate:         // Check if the max
                         accumulate node has been executed on the other threads by
                         adding the wcet
36                          if min_value > thread.time + max_accumulate_node.wcet:
                             // Choose a thread from the set of accepted threads
                             which has min time, work conserving
37                              min_value = thread.time + max_accumulate_node.wcet
38                              min_thread_id = thread.thread_id
39                              cond = True
40              update_threads(threads, node)     // Update the threads by adding the
                 node to the thread that was selected
41  end
```

Algorithm 16: Update Threads

```
def update_threads(threads, node):
    /* Update thread which includes an addition of a node to the threads  */
    for thread in threads:          // Update the threads with the times that
     includes the node wcet
        if thread.thread_id == min_thread_id:
            if cond:
                thread.time += node.wcet + max_accumulate_node.wcet
            else:
                thread.time += node.wcet
            node.accumulate = thread.time
            node.thread_id = thread.thread_id
    end
end
```

Algorithm 17: Intra-task interference calculation

```
Input: task: DagTask
Output: response_time: int
def calculate_intra_task_interference(task:  DagTask) :  int:
    /* The function computes the intra-task interference for a given dag
       with priorities assigned to the subtask                          */
    subtasks = task.dag.topological_sort()    // Using networkx retrieve the nodes
     in a topological order
    source = tasks[0]
    parallelism = [source]
```

```
5
6   while len(parallelism) != 0:       // Pop the highest priority node from
     parallelism
7       parallelism.sort()             // Sort the subtasks according to priorities
8       for node in parallelism:
9           if node.execute:   // Pop the first highest priority node which is
             ready
10              parallelism.remove(node)
11              break
12          else:
13              node = None
14      end
15      heap_push(task, node, threads)      // Push the node to the threads i.e.,
        node executes on a thread
16      if len(parallelism) == 0:  // When a layer exists, create a buffer on
         the threads that do not have max time of threads
17          max_time = max(max_time, threads.time)
18          foreach thread in threadsdo thread.time = max_time
19      for child in neighbors(node):  // Find the ready children of the node
         that was executed
20          ancestors = predecessors(child)       // Check all the ancestors of the
             child
21          add_child = True for ancestor in ancestors: // A child is ready only
             if all its ancestors are visited
22              if !  ancestor.visited:
23                  add_child = False
24                  break
25          end
26          if !  add_child:    // Indicates if a child is ready to execute or
             not
27              child.execute = False
28          else:
29              child.execute = True
30          if child not in parallelism:    // Add all children to parallelism
31              parallelism.append(child)
32      end
33  max_time = max(max_time, thread.time)      // Find the max time of all the
     threads return max_time - critical_path
34 end
```

List of Figures

List of Tables

List of Algorithms

Listings

Glossary

DagTask Class name used in implementation which denotes a task modeled as a DAG. 30, 40, 54–56

fixed-priority scheduler task priorities that do not change at any point during execution, the schedulers schedule first the highest priority task of all those tasks. 4

global One scheduler manages the spatial and temporal dimensions of the scheduling for all cores. 4, 76

layer A layer in a Directed Acyclic Graph (DAG) consist of nodes or subtasks that are ready at the same time in parallel. For instance, for a single source DAG, layer 1 consists of only the source node, in layer 2 consists of all descendants of the first layer. Generally layer n consists of all descendants of all nodes in layer n - 1. 36, 37, 39, 55–57

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming [56]. 1

partitioned Each core is managed by one scheduler which is responsible only for the time scheduling. 76

Acronyms

AOCS Attitude and Orbit Control System. 5

ATON Autonomous Terrain-based Optical Navigation. 2, 8

BIRD Bi-spectral Infrared Detection. 5

DAG Directed Acyclic Graph. iii–v, 2–4, 9–16, 21, 26, 27, 30–33, 36, 37, 40, 44–46, 49–51, 54–57, 60–69, 71–76, 86, 87, 91

NFJ Nested Fork-Join. v, 49–51, 53, 66–69, 73, 86, 87

ODARIS Onboard Data Analysis, Real-time Information System. 8

ScOSA Scalable On-Board Computing for Space Avionics. 8

WCET Worst Case Execution Time. v, 9, 12–15, 31, 37–40, 44, 45, 51, 52, 54, 56, 57, 61, 70, 86, 89

Bibliography

[1] H. Abaza. "Worst-Case Execution Time Analysis for C++ based Real-Time On-Board Software Systems". MA thesis. Technische Universität Hamburg, Apr. 2021. URL: https://elib.dlr.de/141634/.

[2] H. Abaza, Z. A. H. Hammadeh, and D. Lüdtke. "DELOOP: Automatic Flow Facts Computation using Dynamic Symbolic Execution". In: *20th International Workshop on Worst-Case Execution Time Analysis*. Ed. by C. Ballabriga. Vol. 103. OpenAccess Series in Informatics. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 3:1–3:12. URL: https://elib.dlr.de/186919/.

[3] D. S. Aric Hagberg Pieter Swart. *NetworkX*. URL: https://networkx.org/ (visited on 08/25/2022).

[4] D. S. Aric Hagberg Pieter Swart. *NetworkX*. URL: https://networkx.org/documentation/stable/reference/classes/digraph.html (visited on 08/25/2022).

[5] S. Baruah. "The federated scheduling of systems of conditional sporadic DAG tasks". In: *2015 International Conference on Embedded Software (EMSOFT)*. 2015, pp. 1–10. DOI: 10.1109/EMSOFT.2015.7318254.

[6] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela. "The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks". In: *2015 27th Euromicro Conference on Real-Time Systems*. 2015, pp. 222–231. DOI: 10.1109/ECRTS.2015.27.

[7] M. Bertogna, M. Cirinei, and G. Lipari. "Improved schedulability analysis of EDF on multiprocessor platforms". In: *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. 2005, pp. 209–218. DOI: 10.1109/ECRTS.2005.18.

[8] M. Bertogna and M. Cirinei. "Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 149–160. DOI: 10.1109/RTSS.2007.31.

[9] E. Bini and G. Buttazzo. "Biasing effects in schedulability measures". In: *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* 2004, pp. 196–203. DOI: 10.1109/EMRTS.2004.1311021.

[10] E. Bini and G. C. Buttazzo. "Measuring the Performance of Schedulability Tests". In: *Real-Time Syst.* 30.1–2 (May 2005), pp. 129–154. ISSN: 0922-6443. DOI: 10.1007/s11241-005-0507-9. URL: https://doi.org/10.1007/s11241-005-0507-9.

[11] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin. "Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms". In: *2013 25th Euromicro Conference on Real-Time Systems*. 2013, pp. 25–34. DOI: 10.1109/ECRTS.2013.14.

[12] I. of Computer Network Engineering at TU Braunschweig (IDA-TUBS). *pyCPA: Compositional Performance Analysis in Python*. URL: https://pycpa.readthedocs.io/en/latest/ (visited on 08/25/2022).

[13] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. "Random Graph Generation for Scheduling Simulations". In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. SIMUTools '10. Torremolinos, Malaga, Spain: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010. ISBN: 9789639799875.

[14] M. Coutinho, J. Rufino, and C. Almeida. "Response Time Analysis of Asynchronous Periodic and Sporadic Tasks Sheduled by a Fixed Priority Preemptive Algorithm". In: *2008 Euromicro Conference on Real-Time Systems*. 2008, pp. 156–167. DOI: 10.1109/ECRTS.2008.30.

[15] P. Erdös and A. Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[16] J. Fonseca, G. Nelissen, and V. Nélis. "Improved Response Time Analysis of Sporadic DAG Tasks for Global FP Scheduling". In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. RTNS '17. Grenoble, France: Association for Computing Machinery, 2017, pp. 28–37. ISBN: 9781450352864. DOI: 10.1145/3139258.3139288. URL: https://doi.org/10.1145/3139258.3139288.

[17] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201575949.

[18] P. S. Foundation. *Heap queue algorithm*. URL: https://docs.python.org/3/library/heapq.html (visited on 08/25/2022).

[19] P. S. Foundation. *Python*. URL: https://www.python.org/ (visited on 08/25/2022).

[20] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". In: *SIAM J. Appl. Math.* 17.2 (Mar. 1969), pp. 416–429. ISSN: 0036-1399. DOI: 10.1137/0117039. URL: https://doi.org/10.1137/0117039.

[21] Z. A. Haj Hammadeh. "Deadline Miss Models for Temporarily Overloaded Systems". PhD thesis. Sept. 2019. DOI: 10.24355/dbbs.084-201909020857-0. URL: http://uri.gbv.de/document/opac-de-84:ppn:1678285706.

[22] Z. A. H. Hammadeh, T. Franz, O. Maibaum, A. Gerndt, and D. Lüdtke. "Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems". In: *15th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*. July 2019, pp. 29–34. URL: https://elib.dlr.de/128249/.

[23] M. Han, N. Guan, J. Sun, Q. He, Q. Deng, and W. Liu. "Response Time Bounds for Typed DAG Parallel Tasks on Heterogeneous Multi-Cores". In: *IEEE Transactions on Parallel and Distributed Systems* 30.11 (2019), pp. 2567–2581. DOI: 10.1109/TPDS.2019.2916696.

[24] Q. He, M. Lv, and N. Guan. "Response Time Bounds for DAG Tasks with Arbitrary Intra-Task Priority Assignment". In: *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Ed. by B. B. Brandenburg. Vol. 196. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 8:1–8:21. ISBN: 978-3-95977-192-4. DOI: 10.4230/LIPIcs.ECRTS.2021.8. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13939.

[25] Q. He, x. jiang xu, N. Guan, and Z. Guo. "Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores". In: *IEEE Transactions on Parallel and Distributed Systems* 30.10 (2019), pp. 2283–2295. DOI: 10.1109/TPDS.2019.2910525.

[26] X. He and Y. Yesha. "Parallel recognition and decomposition of two terminal series parallel graphs". In: *Information and Computation* 75.1 (1987), pp. 15–38. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(87)90061-7. URL: https://www.sciencedirect.com/science/article/pii/0890540187900617.

[27] D. Isovic. "Handling sporadic tasks in real-time systems : Combined offline and online approach". In: 2001.

[28] K. Jeffay. "Scheduling sporadic tasks with shared resources in hard-real-time systems". In: *[1992] Proceedings Real-Time Systems Symposium*. 1992, pp. 89–99. DOI: 10.1109/REAL.1992.242673.

[29] X. Jiang, N. Guan, X. Long, and W. Yi. "Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors". In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. 2017, pp. 80–91. DOI: 10.1109/RTSS.2017.00015.

[30] Kasahara and Narita. "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing". In: *IEEE Transactions on Computers* C-33.11 (1984), pp. 1023–1029. DOI: 10.1109/TC.1984.1676376.

[31] S. Kato and Y. Ishikawa. "Gang EDF Scheduling of Parallel Task Systems". In: *2009 30th IEEE Real-Time Systems Symposium*. 2009, pp. 459–468. DOI: 10.1109/RTSS.2009.42.

[32] P. Kenny, K. Schwenk, D. Herschmann, A. Lund, V. Bansal, Z. A. H. Hammadeh, A. Gerndt, and D. Lüdtke. "Parallelizing On-Board Data Analysis Applications for a Distributed Processing Architecture". In: *2nd European Workshop on On-Board Data Processing (OBDP2021)*. June 2021. URL: https://elib.dlr.de/142860/.

[33] U. Khamdamov and. "A comparative analysis of parallel programming models for C++". In: (Apr. 2019), pp. 295–298.

[34] Y.-K. Kwok and I. Ahmad. "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors". In: *ACM Comput. Surv.* 31.4 (Dec. 1999), pp. 406–471. ISSN: 0360-0300. DOI: 10.1145/344588.344618. URL: https://doi.org/10.1145/344588.344618.

[35] K. Lakshmanan, S. Kato, and R. Rajkumar. "Scheduling Parallel Real-Time Tasks on Multi-core Processors". In: *2010 31st IEEE Real-Time Systems Symposium*. 2010, pp. 259–268. DOI: 10.1109/RTSS.2010.42.

[36] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Vol. 2050. June 2004. ISBN: 3-540-42184-X. DOI: 10.1007/3-540-45318-0.

[37] H. Lee, S. Cho, Y. Jang, J. Lee, and H. Woo. "A Global DAG Task Scheduler Using Deep Reinforcement Learning and Graph Convolution Network". In: *IEEE Access* 9 (2021), pp. 158548–158561. DOI: 10.1109/ACCESS.2021.3130407.

[38] J. Li, K. Agrawal, C. Lu, and C. D. Gill. "Analysis of Global EDF for Parallel Tasks". In: 2013.

[39] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. "Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks". In: *2014 26th Euromicro Conference on Real-Time Systems*. 2014, pp. 85–96. DOI: 10.1109/ECRTS.2014.23.

[40] G. Lipari, G. Buttazzo, and L. Abeni. "A bandwidth reservation algorithm for multi-application systems". In: *Proceedings Fifth International Conference on Real-Time Computing Systems and Applications (Cat. No.98EX236)*. 1998, pp. 77–82. DOI: 10.1109/RTCSA.1998.726354.

[41] R. S. Ltd. *Discover Worst-Case Execution Time*. URL: https://www.rapitasystems.com/worst-case-execution-time (visited on 08/25/2022).

[42] A. Lund, Z. A. H. Hammadeh, P. Kenny, V. Vishav, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke. "ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture". In: *CEAS Space Journal* (Mai 2021). URL: https://elib.dlr.de/142681/.

[43] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho. "Response-Time Analysis of Synchronous Parallel Tasks in Multiprocessor Systems". In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versaille, France: Association for Computing Machinery, 2014, pp. 3–12. ISBN: 9781450327275. DOI: 10.1145/2659787.2659815. URL: https://doi.org/10.1145/2659787.2659815.

[44] O. Maibaum, D. Lüdtke, and A. Gerndt. "Tasking Framework: Parallelization of Computations in Onboard Control Systems". In: *ITG/GI Fachgruppentreffen Betriebssysteme*. http://www.betriebssysteme.org/Aktivitaeten/Treffen/2013-Berlin/Programm/. Nov. 2013. URL: https://elib.dlr.de/87505/.

[45] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. "Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems". In: *2015 27th Euromicro Conference on Real-Time Systems*. 2015, pp. 211–221. DOI: 10.1109/ECRTS.2015.26.

[46] P. Michailidis and K. G. Margaritis. "Scientific computations on multi-core systems using different programming frameworks". In: *Applied Numerical Mathematics* 104 (Jan. 2015). DOI: 10.1016/j.apnum.2014.12.008.

[47] I. NumFOCUS. *NumPy*. URL: https://numpy.org/ (visited on 08/25/2022).

[48] E.-A. Risse, K. Schwenk, H. Benninghoff, and F. Rems. "Guidance, Navigation and Control for Autonomous Close-Range-Rendezvous". In: *Deutscher Luft- und Raumfahrtkongress 2020*. Oktober 2020. URL: https://elib.dlr.de/137654/.

[49] S. Rosengren. "Random Graph and Growth Models". In: 2020.

[50] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. "Multi-core Real-Time Scheduling for Generalized Parallel Task Models". In: *2011 IEEE 32nd Real-Time Systems Symposium*. 2011, pp. 217–226. DOI: 10.1109/RTSS.2011.27.

[51] M. Schmid and J. Mottok. "Response Time Analysis of Parallel Real-Time DAG Tasks Scheduled by Thread Pools". In: *29th International Conference on Real-Time Networks and Systems*. RTNS'2021. NANTES, France: Association for Computing Machinery, 2021, pp. 173–183. ISBN: 9781450390019. DOI: 10.1145/3453417.3453419. URL: https://doi.org/10.1145/3453417.3453419.

[52] M. A. Serrano, A. Melani, M. Bertogna, and E. Quinones. "Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2016, pp. 1066–1071.

[53] M. Staron and D. Durisic. "AUTOSAR Standard". In: *Automotive Software Architectures: An Introduction*. Cham: Springer International Publishing, 2017, pp. 81–116. ISBN: 978-3-319-58610-6. DOI: 10.1007/978-3-319-58610-6_4. URL: https://doi.org/10.1007/978-3-319-58610-6_4.

[54] S. Theil, N. A. Ammann, F. Andert, T. Franz, H. Krüger, H. Lehner, M. Lingenauber, D. Lüdtke, B. Maass, C. Paproth, and J. Wohlfeil. "ATON (Autonomous Terrain-based Optical Navigation) for exploration missions: recent flight test results". In: *CEAS Space Journal* (März 2018). URL: https://elib.dlr.de/119557/.

[55] T.-s. Tia, J. W.-S. Liu, J. Sun, L. Jun, and R. Ha. *A Linear-Time Optimal Acceptance Test for Scheduling of Hard Real-Time Tasks*. 1994.

[56] Wikipedia. *OpenMP*. URL: https://en.wikipedia.org/wiki/OpenMP (visited on 08/25/2022).