CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Fault-tolerant scheduling of real-time parallel DAG tasks on multiprocessors

Estimating response time and processor requirements to examine schedulability of parallel DAG tasks

Master's thesis in Computer science and engineering

Gustaf Bodin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Fault-tolerant scheduling of real-time parallel DAG tasks on multiprocessors

Estimating response time and processor requirements to examine schedulability of parallel DAG tasks

Gustaf Bodin

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Fault-tolerant scheduling of real-time parallel DAG tasks on multiprocessors
Estimating response time and processor requirements to
examine schedulability of parallel DAG tasks
Gustaf Bodin

Supervisor: Risat Pathan, Department of Computer Science and Engineering
Examiner: Jan Jonsson, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Fault-tolerant scheduling of real-time parallel DAG tasks on multiprocessors
Estimating response time and processor requirements to examine schedulability of parallel DAG tasks
Gustaf Bodin
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The imperative of maximizing hardware utilization compels innovation and a strive towards finding more efficient solutions in real-time systems. The directed acyclic graph (DAG) model of parallel tasks is commonly used to represent the data dependencies of real-world applications. Providing techniques for tolerating and recovering from hardware and software faults for such a model is paramount to its usability in critical systems. Sectors like avionics and automotive require fault tolerance to ensure certification and guarantees of safe operation, which is why this combination is important. This thesis considers transient faults. Examining all possibilities of faults occurring in a DAG task is computationally expensive. Therefore, developing efficient methods for bounding the worst-case makespan under faulty conditions is a non-trivial problem and one which is examined in this thesis.

A fault-aware schedulability test for a taskset can be derived from finding the number of processors required to meet each task's deadline in the taskset. This thesis introduces six novel fault-aware schedulability tests that explicitly account for the runtime overhead of using fault recovery through node re-execution. Further, a work-conserving scheduler is assumed and the federated scheduling technique is employed to address the problem of guaranteeing the schedulability of DAG tasksets. To bound the worst-case interference caused by re-executed nodes, the tests employ new analytical techniques and build upon existing fault-unaware scheduling techniques for efficient scheduling of DAG tasks.

To evaluate the effectiveness of the proposed tests, a simulation framework was developed that is capable of generating random DAG tasks whose structure and computational load reflect that of real-world applications. Simulation results indicate that exploiting structural information of multiple long paths of a DAG task significantly enhances the power of the proposed tests in determining schedulability, regardless of the variation in the simulation parameters.

Keywords: real-time scheduling, parallel tasks, fault tolerance, federated scheduling, work-conserving scheduling, computer science

Acknowledgements

I sincerely want to thank my supervisor Risat Pathan, who gave me the opportunity of this thesis and provided invaluable guidance and support, and through his wisdom and extensive knowledge within the field made this work possible. I am grateful to all who highlighted the importance and fascinating nature of real-time systems during my time at Chalmers University of Technology, especially my examiner Jan Jonsson for his engaging real-time systems course.

Gustaf Bodin, Gothenburg, 2025-01-31

Contents

List of Figures

1
Introduction

The imperative of maximizing hardware utilization compels innovation and a strive towards finding more efficient solutions in time-critical systems. The demand for utilization is partly met through the computational power of multicore processors. However, the reliance on such architectures necessitates additional strategies and safety functions for efficient task scheduling, parallel execution, and fault tolerance in software and hardware. Parallel tasks are a new way to make applications exploit parallel multicore architectures. This work aims to answer how faults occurring in a multicore processor environment can be tolerated such that parallel tasks meet their timing constraints.

A parallel task can represented by a directed acyclic graph (DAG), which is made up of nodes interconnected by edges. These parallel tasks are subject to similar constraints as sequentially executed tasks but can leverage the use of multiple processors at once. Multiple nodes can use multiple processor cores simultaneously to distribute the workload of parallel tasks across the available processors.

A limitation of classical sequential task execution is that some processor cores are left unused because the tasks lack enough task-level parallelism and do not exploit the cores to their full potential. For workloads with strict timing constraints, parallel execution can leverage the leftover execution time of idle processors, potentially mitigating missed deadlines or enabling the execution of a larger workload by executing tasks concurrently. The ability to divide applications into smaller segments using a parallel programming paradigm like OpenMP [1] consequently enables intra-task parallelism, and thereby greater exploitation of a parallel architecture [2] [3].

In real-time systems, ensuring correctness in terms of both timing constraints and logical results is vital, as such systems often provide critical functionality. Examples of such systems are prevalent in sectors like avionics, the automotive industry, and industrial control systems, where the certification of correct operation is paramount. In the automotive industry, larger battery capacity and greater power output in electric vehicles enable greater processing capacity within each vehicle. This, in turn, enables the implementation of sophisticated safety systems like object detection for an automatic collision avoidance system where meeting timing constraints like deadlines or periodicity are essential [4].

It is important to consider the occurrence of both temporary and permanent faults in the execution of real-time tasks, since they can have devastating consequences for critical systems. A temporary bit flip in a processor register or a permanent processor failure can jeopardize the function and safety of a system. Understanding and modeling real-time systems, including the expected faults, is therefore key to the success of the system, especially in safety-critical systems where human

lives may be at stake. Extending the offline analysis to consider how many faults can occur, how faults can occur, and what type of faults to expect is required to provision a system to tolerate them. The intersection of fault tolerance and parallel task execution is highly relevant as computing systems strive to meet the demands of both reliability and high-performance computation.

Validating system behavior under a provided set of assumptions, which includes potential faults for fault-tolerant systems, is typically done using offline analysis. Such analysis involves analyzing a real-time system's behavior before it is deployed and focuses on verifying that the task deadlines of the system are met even in the presence of faults. System designers thus need to utilize offline analysis to verify that all timing constraints are met prior to the operation of the system [2], whilst considering the allocation of resources dedicated to the recovery or masking of faults.

Embracing parallelism allows for a more dynamic allocation of resources, further exploiting the full potential of modern multiprocessor platforms. By leveraging architectures designed for concurrency, and enabling parallel execution, parallel tasks can potentially be completed faster. Therefore, it provides better chances of meeting task deadlines, than if the same tasks were to execute sequentially on a single processor. OpenMP is an API that allows for multiprocessing programming in common languages like C and C++, enabling parallel execution on many modern platforms wherein programmers explicitly specify the actions to be taken by the compiler and runtime system [1]. One realization of parallel tasks on a multiprocessor architecture can be done through the use of a DAG. DAGs provide a general way of modeling parallel tasks[2][5]. A multithreaded runtime system called Cilk [6] supports the DAG model, wherein the computations within Cilk can be viewed as DAGs.

A DAG task is made up of a set of nodes, where the nodes represent sequential execution units, and edges represent the data or other dependency constraints between them. Figure 1.1 exemplifies a DAG task.

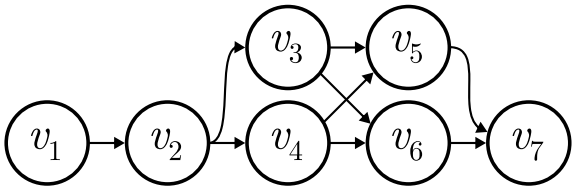


Figure 1.1: An example representation of a DAG task made up of a set of nodes

Software bugs and hardware problems can lead to incorrect calculations, or limit a system's capacity to provide its intended service. Therefore, error detection is paramount in critical systems for detecting erroneous calculations and errors due to processor failures. The earlier an error can be detected, the sooner a recovery mechanism can be employed to minimize or negate its impact. For sequential tasks, error detection typically occurs at the end of their execution, for example, comparing the results with a known correct execution, if no additional mechanism is employed for earlier detection. For parallel DAG tasks, errors are instead typically detected at the node level, i.e. after a node completes executing. Node-level error detection enhances efficiency and reduces a system's potential worst-case workload (WCW) compared to detecting errors at the end of the execution of a task. For example,

detecting errors in individual nodes could allow for earlier termination of computations and the unused remaining time could be used for selective re-execution of the erroneous nodes that generate errors.

Note that, safety-critical systems typically face practical resource constraints such as space, weight, and power (SWaP)[7]. Consequently, SWaP is highly relevant when addressing fault tolerance, where handling faults creates tension between conserving resources and the ability to efficiently handle faults.

1.1 Contributions

In this thesis, the problem of fault-tolerant scheduling of parallel DAG tasksets is considered. The tasks are assumed to suffer multiple transient faults. The work aims to answer how faults can be tolerated for a certain system model, such that it ensures all task deadlines are met. To solve this problem, we will propose schedulability tests in the form of polynomial time complexity algorithms. Then, experiments are run in a simulation environment to compare the efficacy of the work. The contributions of this thesis can be outlined as follows, where we will:

- Propose mechanisms to implement fault tolerance in parallel DAG tasks on a multiprocessor platform. To the best of our knowledge, this has not been proposed before.
- Propose a model to account for transient faults for parallel DAG tasks. To the best of our knowledge, this combination is novel.
- Propose a scheduling algorithm capable of accounting for fault tolerance in terms of transient faults. The main method for fault recovery is done through re-execution, and the focus lies in utilizing existing and well-established scheduling algorithms to introduce fault tolerance capability, as the scheduling algorithm itself is not the focus of the work.
- Propose analysis of the scheduling algorithm for both single and multiple DAG tasks, comparing different analysis methods with the goal of minimizing pessimism. Here, pessimism refers to the overestimation of the algorithms' results.
- Perform experimental analysis using randomly generated tasksets to evaluate our proposed scheduling algorithm and analysis. We want to determine how sensitive the schedulability tests are in terms of determining the schedulability of randomly generated tasksets.

1.2 Thesis Outlook

This thesis is organized as follows:

Chapter 1: *Introduction* introduces the problem area and the importance of the work of this thesis.

Chapter 2: *Background* provides fundamental theoretical concepts for and provides the assumed system model, as well as contextualizes this work with prior research in the topic area.

Chapter 3: *Problem Formulation* formulates the problem this thesis aims to answer and provides limiting factors.

Chapter 4: *Algorithms and Analysis* presents a progression of the fault-tolerant algorithms and analysis this work provides, beginning with examples and relevant background for understanding parallel DAG task analysis with fault tolerance.

Chapter 5: *Evaluation* describes the simulation setup and DAG task generation, and explains how it is employed for the analysis presented in Chapter 4. Then, it presents the results of this work, examining schedulability for a single DAG task and then tasksets. Further, it provides several examples of varying system parameters and discusses intuitions and reasons for the outcome of the results.

Chapter 6: *Conclusion and Future Work* highlights key takeaways of the work, and provides the conclusion to the work.

2

Background

This chapter introduces the foundational concepts for the research presented in this work. Section 2.1 begins by outlining the fundamental principles of real-time scheduling, including task characteristics, terminology, scheduling techniques, and common scheduling algorithms and paradigms. In Section 2.2, the system model is characterized and presented, covering the task model, fault model, and processor model. The system model contextualizes the expected system environment and the concept of fault tolerance and helps introduce the problem area to be solved in this work. Section 2.3 highlights previous work in the related areas of parallel DAG scheduling, fault tolerance, and federated scheduling.

2.1 Real-Time Systems: Key Concepts

The sequential task model is presented in this section, but soon the parallel task model will be introduced as well. The commonly used term *task* within the real-time systems field is a resource-sharing unit of work, typically some code or program, to be executed within a particular system. Real-time software is generally modeled using a collection of tasks. A task *arrival* (or *release*) denotes a time at which the task is ready to be scheduled. A recurrent task G is typically defined by a set of parameters $G = (C, T, D)$. The parameter C denotes the assumed *worst-case execution time* (WCET), and the period T is the minimum time between two successive arrivals, and D is the task's deadline relative to its arrival, i.e. the time before which the task should complete its execution.

The WCET of a task is the maximum expected time the task takes to execute on a specific processor platform. A deadline is considered to *constrained* if $D \leq T$, where constrained deadlines will be considered for the scheduling of multiple DAG tasks in this work. Instead, *implicit* deadlines are when the deadline is strictly equal to the task's period, $D = T$, and will be considered for single DAG task scheduling. Figure 2.1 exemplifies the timing relationship between the fundamental parameters of a task. The deadline D is relative to the task's arrival, and the WCET C is an upper bound estimating the execution time of the task in the worst case. The upward and downward arrows represent the task's release time and deadline, respectively.

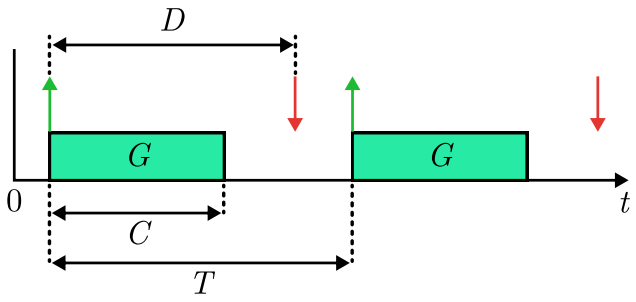


Figure 2.1: Timing diagram of the fundamental parameters of a task G

The *scheduling* of multiple tasks within a system is a policy of strategic allocation of the finite CPU time available, for which the tasks compete [7]. If the tasks have stringent timing conditions, i.e. it is critical for programs or functions to be completed on time, it is known as a *hard* real-time system. A scheduling algorithm is considered work-conserving if it ensures keeping all available resources, i.e. processor cores, busy whenever eligible tasks are waiting for execution. This work will assume work-conserving scheduling for the subsequent analysis, which will attempt to address the problem of fault-tolerant scheduling of a collection of parallel DAG tasks.

In real-time systems, a scheduler is responsible for deciding which task is allowed to execute on the given processor platform. In fixed-priority (FP) scheduling, the scheduler uses a predefined priority ordering of the tasks to determine the execution order. It ensures that the highest priority task is executed first out of tasks ready for execution. Earliest deadline first (EDF) is a dynamic priority scheduling algorithm that assigns priority to the tasks dynamically during runtime. For all scheduling decisions under EDF, the task with the earliest absolute deadline will be treated as the highest priority, and thus first to be scheduled out of the tasks ready for execution [8].

Preemptive scheduling is a technique in which higher-priority tasks are allowed to preempt, i.e. temporarily halt, the execution of lower-priority tasks, seizing the execution time of the processor. A preempted task can continue execution once the higher-priority task has finished. Preemption will not be considered in this work. This is because, as we will discuss shortly, when a node is dispatched for execution, it will continue executing until it completes. The node of one DAG will execute on a dedicated set of processors and never preempts the nodes of another DAG that has another set of dedicated processors.

When a task G is recurrent there will be an infinite number of instances of G that will arrive in the system, separated by at least T time units. Each such instance will have an arrival time and a finishing time. The finishing time includes the potential interference from other tasks. The response time of G is the maximum time any instance takes between its arrival time and its finishing time. The *makespan* of a DAG task denotes its response time [2], and will be used interchangeably.

A task is not guaranteed to start execution on its arrival due to it potentially being delayed by the execution of tasks with higher priority, such that no processor is left idle. Figure 2.2 highlights the response time of an example task G, which suffers interference from other tasks after its release on a processor m. The task

meets its deadline, as the response time R does not exceed its deadline.

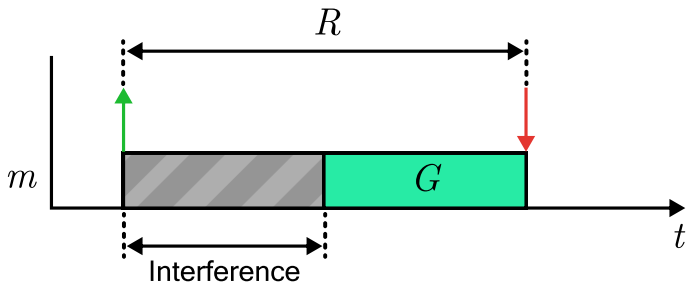


Figure 2.2: Timing diagram showcasing the response time of a task G in relation to its release time

The worst-case response time (WCRT) of a particular task is the maximum time required for that task to complete its execution under a given scheduling algorithm. Similarly, the WCRT of a DAG task is the maximum time required under a given scheduling algorithm to complete the execution for all the nodes. If the WCRT for all tasks in a taskset is less than or equal to their deadlines under some scheduling algorithm, then the taskset is referred to as *schedulable*.

A schedulability test can be used to examine if a taskset is schedulable or not on a given hardware platform and fault model. Given a taskset and a scheduling algorithm, a schedulability test determines if all the tasks of the taskset will meet their deadline on a given platform. The tests can be categorized into sufficient, necessary, and exact. A schedulability test is sufficient if a positive outcome indicates the task or taskset is schedulable. If the test fails, no conclusion can be drawn about schedulability. In contrast, a necessary schedulability test indicates that the task or taskset is unschedulable if the test produces a negative outcome, and no conclusion can be drawn from a positive outcome. Finally, an exact test is both sufficient and necessary.

Real-time task scheduling on multiprocessor architectures is commonly based on either the *global* or the *partitioned* approach [9][10]. Task migration is when tasks are allowed to execute on any processor, at any time, even after preemption. Global scheduling allows task migration, therefore enabling the movement of tasks between processors. In contrast, partitioned scheduling instead limits each task to only execute on a preassigned processor, decided before running the system. Each processor executes some uniprocessor scheduling algorithm, e.g. EDF, rate-monotonic (RM), or deadline-monotonic (DM) scheduling.

A common scheduling algorithm for DAG tasksets is that of *federated scheduling*, which offers high flexibility and good real-time performance [11]. In federated scheduling, tasks are classified into *heavy* tasks and *light* tasks based on the utilization of each task. Task utilization is the ratio between WCET and its period. Tasks with utilization greater than 1 are considered to be heavy, whilst tasks with utilization less than or equal to 1 are considered light. All heavy tasks are assigned a set of dedicated processors. The light tasks follow partitioned scheduling and are assigned to the remaining processors because they can be executed sequentially.

Federated scheduling can be beneficial in terms of fault tolerance due to the

distribution of heavy tasks across processors. Under federated scheduling, a fault and its subsequent recovery affecting one task will not affect tasks assigned to other processors. Under global scheduling, a fault occurring in one task can affect the schedule of other tasks. In this thesis, federated scheduling will be employed when extending single-task scheduling to allow the scheduling of multiple tasks.

2.2 System Model

Critical systems require verification, and to perform the verification of real-time systems under a certain scheduling algorithm, the model of the real system needs to be considered, starting from software and hardware. To that end, this section presents the task model, the fault model, and the processor model considered in this work.

2.2.1 Task model

The parallel tasks considered in this work are sporadic and are to be modeled as a DAG. Sporadic tasks are infinite sequences of task arrivals, where the minimum inter-arrival time, T, of consecutive arrivals is known as the period of the DAG task. In this thesis, only constrained deadlines ($D \leq T$) are considered. The assumptions of this parallel task model are based on the work of Baruah et al. [5], who provided a generalization of representing parallel sporadic precedence-constrained tasks on a multiprocessor platform. The terms *node* and *vertex* can come to be used interchangeably. A DAG task representation is assumed which resembles that presented by Pathan et al. [2], which characterizes each DAG task by four parameters $G := (T, D, V, E)$, where:

- The set of sequential nodes for a single DAG task is denoted $V = \{v_1, v_2, ..., v_k\}$, where k is the number of nodes in V, i.e. $k = |V|$.
- The set of directed edges, E, is a pairing of consecutive vertices, such that $E \subseteq (V \times V)$. An edge $(u, v) \in E$ defines u as a *successor* of v, and v a *predecessor* of u [12]. The nodes in V will be subject to the precedence constraints of the edges in E.
- Considering the scheduling of multiple tasks, a set of DAG tasks is denoted Γ, and the set of nodes for a task $G_i \in \Gamma$ is instead denoted $V_i = \{v_{i,1}, v_{i,2}, ..., v_{i,k}\}$. The term n commonly denotes the size of Γ, i.e. $n = |\Gamma|$.

A node of a DAG task is a sequential piece of code or program, and an edge (u, v) in E provides a precedence constraint between two nodes, meaning v cannot execute before u has finished. The relative deadline D is the latest time at which the execution of all nodes must have been completed. A successor node can only start executing once all of its predecessors have completed execution, as shown in Figure 2.3, which exemplifies a simple DAG task where node $v_{1,8}$ can only execute once $v_{1,6}$ and $v_{1,7}$ have completed.

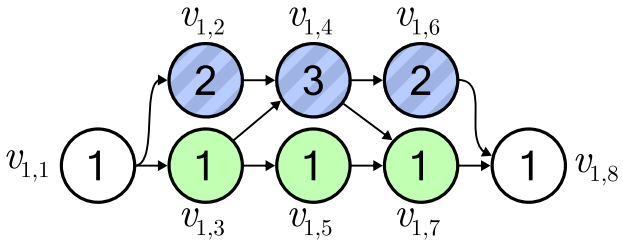


Figure 2.3: A sample DAG task G_1. The number in each node represents the WCET of each node

A node with no incoming edges is known as a *source node*, and a node with no outgoing edges is known as a *sink node*, denoted v_{src} and v_{sink} respectively. Each node $v_{i,j} \in V_i$ of a task G_i, represents some sequentially executed workload with a WCET of $c_{i,j}$.

In a DAG task G_i, a *path* is a series of consecutive (connected) nodes, denoted $\lambda = (\pi_{i,a}, ..., \pi_{i,b})$ where $\forall j \in [a, b-1] : (\pi_{i,j}, \pi_{i,j+1}) \in E_i$. A *complete path* is one where $\pi_a = v_{src}$ and $\pi_b = v_{sink}$, i.e. a path that starts from a node with no incoming edges and ends in a node with no outgoing edges. In Figure 2.3 there are four complete paths. Further, the length of a path λ is the sum of the WCETs of its nodes, i.e. $\text{len}(\lambda) := \sum_{j=a}^{b} c_{i,j}$. The length of the longest path of a task is referred to as L_i.

A DAG task can have multiple longest paths if they are of equal length. A necessary condition for the schedulability of a task G is $L_i \leq D$, i.e. the deadline has to be greater or equal to all paths in the DAG, otherwise, it will miss its deadline. In Figure 2.3, the longest path $L_i = 1 + 2 + 3 + 2 + 1 = 9$. Further, let R be the response time of G, then another necessary condition for a task to be schedulable is $R \leq D$. The time between a task's arrival and its completed execution, including interference, has to be smaller than the deadline for it to be schedulable.

A scheduling algorithm will determine in which order and which vertices of a DAG task will execute at every point in time. An *execution sequence* describes one possible way the nodes can be executed in the system, having a certain ordering and processor assignment. It describes on which processor the nodes execute at every point in time, and is unknown in offline analysis [13], i.e. it is not determined before running the system. Figure 2.4 highlights an example execution sequence (right) of a DAG task (left) with 8 vertices scheduled on two processors.

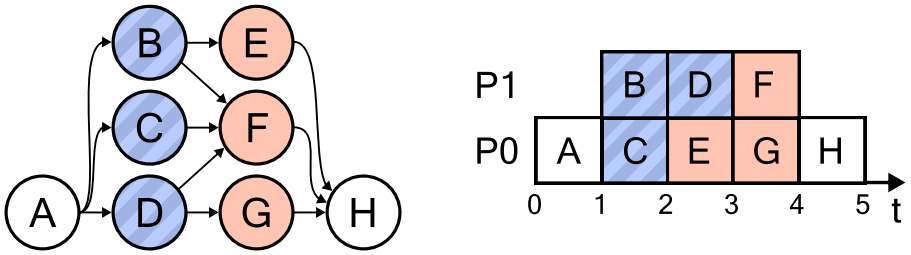


Figure 2.4: Left: Example DAG task assuming a WCET of 1 for all nodes; Right: A specific example execution sequence of the task

Intra-task interference is the interference that nodes within a single DAG can inflict on each other, and this occurs when a node is ready but no processor is available. Figure 2.4 highlights intra-task interference, where node D cannot be executed at the same time as node B and node C, even though it is ready. Figure 2.4 highlights one possible way of scheduling the nodes of the DAG. Another execution sequence can be found by instead prioritizing node D over either node B or node C at $t = 1$. As the number of nodes in a DAG task increases, so does the number of potential execution sequences.

The critical path of a task is the path dictating its response time, i.e. it is the path of nodes during runtime which completes last.

Definition 1. *The critical path λ^* in a task is the path that suffers the most interference for a specific execution sequence. The length of the critical path is defined as L^*.*

When the response time of a task equals the length of its critical path L^*, then the critical path λ^* suffers no interference from nodes not in λ^*. If λ^* suffers any interference, then the response time of the task would increase. The nodes in the critical path are referred to as the *critical nodes*. The critical path for a specific execution sequence can be found by starting at the sink node v_{sink} and iteratively examining which of the nodes predecessors took the longest to complete execution, repeating this step up the DAG until we reach v_{src}.

Definition 2. *The fault-free worst-case workload W_i of an instance of a DAG task G_i is the maximum time required to execute G_i on a single-core processor.*

The workload of a task in the worst case assumes all nodes take their full WCET to complete. The worst-case workload W_i of a task G_i is found by $W_i = \sum_{j=1}^{n} c_{i,j}$. Using W_i we define the fault-free utilization of G_i, and the total fault-free utilization of a taskset Γ as:

$$U_i = \frac{W_i}{T_i}, \qquad U_\Gamma = \sum_{i=1}^{n} U_i \tag{2.1}$$

A necessary condition for schedulability of a taskset Γ is that the total utilization is less than or equal to the number of available processors: $U_\Gamma \leq m$. The utilization represents the ratio between the required execution time within a certain time interval. For example, if the utilization of a task G_i is $U_i = 1.2$ and $D_i = T_i$, it implies the task requires at least $\lceil U_i \rceil = \lceil 1.2 \rceil = 2$ processors in order not to miss its deadline. The ceiling function is required since it is assumed that partial processors cannot be provided.

2.2.2 Fault model

The system's limitations and design are shaped by the assumptions made about the system environment. To achieve fault tolerance, the design of the system must consider a fault model which is about assuming how faults can occur, the frequency of faults, and their potential impact on the system. These assumptions formulate

the *fault model* of the system. The fault model will vary depending on the system as the operating environments are not the same, where the fault model of an electrical vehicle on the road is different than that of an airplane [7].

Guaranteeing the schedulability of a fault-tolerant system assumes a certain fault model [14]. As long as the system model, which includes the fault model, represents the actual operating environment, the scheduling guarantees of meeting task deadlines in the analysis will also hold for the system in operation. However, there will always be some pessimism introduced by the inability to perfectly model reality.

Characterizing and anticipating the types of faults and their effect on the system is important to provide an effective fault-tolerant system. Faults can occur in both software and hardware, and they are based on persistence classified into *permanent*, *intermittent*, and *transient* [7]. This work considers only faults of a transient nature in both hardware and software. However, permanent faults can accounted for by treating them as transient faults and assuming one less available processor.

Hardware permanent faults place the system in a stable erroneous state, for example in case of a permanent processor failure. A failed processor will remain in that state for the entire lifetime of the system. Circumventing such faults requires hardware (spatial) redundancy. On the other hand, hardware transient faults are ones that temporarily place the system in an incorrect state, caused by some malfunctioning component in the system, i.e. a temporary bit flip.

Software faults (also called bugs) can also be categorized into permanent and transient. However, all software faults are permanent and are instead categorized by how they are manifested into errors [7]. Examples of permanent software bugs are ones that always manifest in the execution of a program, i.e. a programmer error that always impacts the result of a program. Instead, if the software fault does not always occur, then it is denoted as transient. These kinds of faults could be ones hiding within conditional execution sequences and can be dependent on external input, which makes them harder to detect.

The fault-tolerant scheduling algorithms of this work consider f transient faults. Let D_{max} denote the maximum relative deadline of any task in a taskset. It is assumed that the number of faults that can occur in the system is limited to at most f within an interval of D_{max}. For any interval smaller than D_{max}, e.g. the deadline of another task in a taskset, the number of faults that can occur is limited to at most f. Therefore, if each of the instances of each task could tolerate f faults without violating the deadline, it can be guaranteed that all the deadlines are met, even if faults occur according to our assumed fault model.

Active and passive backup tasks can be used to counteract the effects of faults. An active backup task always executes in parallel and is a duplicate of the primary execution of the task. If an error is detected in the primary execution of a task, the output of an active backup can be used instead. Similarly, if the output of a task or application is critical, it can be replicated across multiple systems and processors. Actively comparing the replicated tasks' outputs can then be used to detect disagreements, which would suggest an incorrect calculation. In contrast, passive backups only execute if the primary fails. Only passive backup is to be considered in this thesis.

A system cannot tolerate faults without techniques to detect the errors created by them [7]. Therefore, it is important to have reliant mechanisms to detect errors. One such mechanism is software-level tests through unit tests, automated testing, or manual tests. Assertion statements can be used to verify acceptable parameter values and can help ensure the expected outcomes of code-level computations are met at different stages of running a program.

While eliminating bugs during development improves system reliability in general, it does not guarantee fault-free execution, as unexpected errors may still occur at runtime. Therefore, runtime error detection is needed to ensure system dependability. This work will assume a 100% detection rate of errors caused by faults, and also fine-grain error detection. Fine-grain error detection means the faults are detected per node level, rather than at the end of the entire DAG task execution. Error detection can take additional time, but the additional overhead is not modeled, and therefore the time for detection is not accounted for in this thesis. If the error detection coverage is lower than 100%, then system-level fault tolerance is needed, which is an issue not addressed in this thesis.

In system-level fault tolerance, the system hardware can be replicated across three or more system setups, where results are compared and selected with a voting mechanism. The mechanism of utilizing three replica systems for voting is known as triple modular redundancy (TMR) [15]. The effect of faults in one system is negated if the two remaining systems share a single result. Faults in a single system will be ignored if the two remaining systems both share a single result for a computation.

Given a fault model, there arises a need for backup in the real-time tasks of the system, in other words, necessitating redundancy. Consequently, understanding the resulting additional workload due to fault tolerance becomes important. The *worst-case fault occurrence* (WCFO) is the state of faults occurring in the system such that the workload is maximized[16][7]. Meeting the deadlines of a given taskset when anticipating the maximum computational demands of the system provides full assurance in ensuring the correct operation of the system.

Since faults are assumed to occur in the system, the fault-free worst-case workload W_i needs to be extended to account for faults.

Definition 3. *The worst-case workload of a task G_i suffering the maximum number of assumed transient faults, f, is denoted W_{max}^f. We will shortly show how to find W_{max}^f, as there are challenges like the WCFO to be considered.*

The fault model assumed hardware and software transient faults when devising the fault-tolerant scheduling algorithms of this work. Further, each temporary transient error caused by a hardware transient fault was assumed to affect the execution of one node of a DAG task. Consequently, a simple re-execution of such nodes is sufficient to tolerate such faults [17], because transient faults are assumed not to affect the node in the same way if re-executed. However, a node suffering multiple faults may happen if the system is in an environment where the frequency of faults is higher. Re-execution is also appropriate in the case of software transient faults, since they are non-persistent. Re-executing a task will take additional execution time and is referred to as temporal redundancy, whereas spatial redundancy is executing a replica of a task on different redundant hardware. Allocating additional

resources to achieve fault tolerance is not always feasible, which is why this is a resource allocation problem, since we cannot always provide replicas or duplicate execution of tasks while meeting deadlines.

Software permanent faults will not be considered in the fault model, since tolerating these faults requires a strategy different from that of simple re-execution. Therefore, the scheduling algorithms of this work will consider the execution of the original node as the primary method of passive backup for the nodes of the DAG tasks. Hardware permanent faults (processor failures) cannot happen more than once per processor, but they can occur at any time during the lifetime of the system. A node can suffer multiple hardware permanent faults. However, only transient faults are to be accounted for in the analysis of this thesis. Accounting for permanent faults can be done similarly to handling transient faults as follows. Once a processor failure has been detected, it can be assumed that a transient fault occurred, applying the same strategies for tolerating a transient fault. However, the number of available processors has now been reduced by one.

2.2.3 Processor model

The processor model describes the assumptions made about the processor architecture and its specifications. Modern multicore processor architectures may be created with different types of processors, where some cores may target high clock speeds and others power efficiency. A homogeneous processor model is one where all processors are of the same type, and provide the same estimations of WCET for the same task under normal operation.

This thesis will consider m homogeneous processors, each executing with a normalized speed of 1. A node with WCET of c executing on a processor with speed of 1 will take c time units to complete.

2.3 Previous Work

Computing the response time of a parallel task on m processors is an NP-hard problem. There are many aspects affecting the scheduling of the nodes of a DAG task, e.g. nodes may complete execution before their WCET, or faults may hinder their completion. The many interleavings of different nodes and the outcome of scheduling decisions exacerbate the system's predictability. Considering all possibilities in the worst case is not possible, which necessitates a level of pessimism in the schedulability analysis in an attempt to include the consideration of worst-case scenarios. In this thesis, the pessimism is manifested in over-estimations of task parameters such as response time. For the estimations to be considered safe, they need to be greater than or equal to their true value.

A continuous effort within real-time scheduling is that of providing tighter estimates. As estimation of certain parameters is required, a goal is for the estimates to be as close to the real values as possible, i.e. a closer estimate is tighter. Providing tighter estimates can lessen processor overhead and therefore increase the potential efficiency of the system. One such advancement is that of Graham's bound [18], which is a well-known response time bound for DAG tasks assuming work-conserving

scheduling[13]:

$$R \leq L + \frac{W - L}{m} \tag{2.2}$$

2.3.1 Parallel DAG model

A previous study that especially highlights the importance of examining worst-case workload is that of Melani et al. [3]. They utilize a new task model that introduces conditional branches through conditional nodes, which generalizes the standard sporadic DAG model. A conditional DAG differs from regular sporadic DAG in that some nodes are conditional [19]. Informally, conditional nodes represent boolean expressions that are determined during runtime of the application. Based on the outcome, one of two branches from the conditional node will be considered, where the two branches subsequently meet again in a common successor. Conditional DAGs provide a way of modeling applications in which conditional branches differ in length, which can greatly affect the resulting parameters of a DAG (such as response time). An advantageous element of the work of Melani et al. [3] is that their schedulability analysis only requires the worst-case workload and the longest path of the DAG to characterize the conditional structure of the tasks. Further, Melani et al. provide algorithms for deriving these two parameters for the conditional DAG task model. The addition of conditional branches is exploited to provide tighter estimates of interference.

In a recent work by He et al. [13] a closed-form bound on the response time of DAG tasks is proposed. This was done using the total workload and multiple long paths of a DAG task. In contrast, Graham's bound only considers the single longest path of the DAG. He et al. [13] also provide an extension to the scheduling of multiple DAG tasks, where they use a mix of federated scheduling and global EDF or partitioned EDF. They also provide a schedulability test assuming their new response time bound. They first consider work-conserving single DAG analysis to examine the schedulability of a task. The idea is to allocate a processor to exclusively execute the longest path, and then apply Graham's bound to examine the response time of the remaining DAG structure. This process can then be repeated iteratively by excluding another path from the task, and considering one less processor. Using multiple long paths has the potential of reducing the required number of processors required for the execution of DAG tasks, i.e. providing a less pessimistic analysis.

Commonly, the scheduling of parallel tasks considers physical processors, however, Jiang et al. [11] formulate the idea of constructing different types of *virtual processors* on the physical processors, denoted *active-VP* and *passive-VP* respectively. Leftover processing capacity from active-VPs is allocated to other tasks as passive-VPs, where passive-VPs execute with low priority and active-VPs execute with high priority. A primary reason behind using virtual processors is to achieve a higher level of control over the remaining processing capacity left over to lower-priority tasks, thus increasing the potential of guaranteeing schedulability for tasks with low priority, and problems similar to those suffered by federated scheduling. In [11] they provide schedulability tests for DAG tasks on passive-VP groups, and DAG tasks in the general case of utilizing both a passive-VP and active-VP group.

In an attempt to improve the schedulability of DAG tasks, Jiang et al. [12]

showcase that task decomposition strategies emerge as a potential approach for mitigating workload irregularities. The decomposition-based scheduling paradigm allows DAG tasks to be transformed into a set of independent sequential tasks by introducing artificial release times and deadlines for the workload within the task's vertices [12]. Such transformation of the DAG model is performed such that it does not break the interdependencies between connected vertices, as long as the sporadic task executes between its artificial release and deadline. However, it should be noted that the schedulability of the resulting tasks could be worsened compared to the original, due to the shorter relative deadline of the tasks. This has motivated recent studies to attempt to find general and efficient decomposition-based techniques, such as the work done by Jiang et al. [12]. If L_i denotes the longest path, then the *laxity* of a task G_i is defined as $D_i - L_i$, which represents the leftover execution time not spent executing L_i until the next release of G_i.

The primary concept of the decomposition-based techniques of Jiang et al.[12] is to distribute task laxity over the segments created through the artificial release and deadlines of task vertices. This distribution of laxity is done to make the segment's workload and its maximal density of the contained vertices as uniform as possible, to minimize the load and minimize the maximal density among all vertices.

2.3.2 Federated scheduling

Given its strengths, federated scheduling still suffers significant resource wasting, which prior work [20][21] attempts to reduce; successfully to a limited extent. Federated scheduling may not fully utilize all processors that are dedicated exclusively to tasks all of the time. This can lead to idle processors for portions of the task execution, reducing overall system efficiency. Another problem with federated scheduling is when dedicating processors to tasks. For example, a task may require two full processors and a partial third. Therefore, when allocating more processors than a task can effectively utilize, it leads to wasted resources and limits the ability to efficiently distribute resources for other tasks.

2.3.3 Fault tolerance

Fault tolerance is a vital aspect to consider when designing robust and reliable safety- and mission-critical systems. Several techniques for handling faults have been proposed and studied in the context of real-time systems. A common goal is finding resource-efficient scheduling algorithms in an attempt to minimize the additional workload introduced as a consequence of executing backups for tolerating faults, whilst providing mechanisms for recovering from the produced errors. Often prior work considers fault tolerance in the context of either global or partitioned scheduling of tasks on multiprocessor architectures.

A work considering fault tolerance for global scheduling on a multiprocessor is that of Pathan [22] who proposed a global resource-efficient fault-tolerant scheduling algorithm called `FTM`. The `FTM` algorithm uses additional backup executions of tasks to recover from errors caused by transient and hardware permanent faults by scheduling the backups during runtime, therefore not considering backups during

the offline allocation of resources. Both permanent and non-permanent faults are viewed as job errors because re-execution is performed in both cases. The only difference is that one less processor is considered in the case of permanent processor failure. Permanent processor failures caused by hardware permanent faults can be regarded as job errors since it is solved by scheduling the backups of the affected jobs on any non-erroneous processor.

Another work by Pathan and Jonsson [14] presents the schedulability analysis of a fault-tolerant scheduling algorithm for uniprocessors, denoted $\texttt{FPS}_{\wp}$. This algorithm assumes fixed-priority task assignment and a uniprocessor, and they provide an exact feasibility test for fixed-priority scheduling of sporadic tasksets while tolerating multiple hardware transient faults. The WCFO was considered using dynamic programming. Further, Pathan and Jonsson [14] explain how the feasibility of a task is dependent on the interference of higher priority tasks, together with the faults affecting the higher priority tasks. The maximum amount of execution required in the interval $[0, D_i]$ for a task G_i is an important metric to examine. Therefore, a similar analysis will be utilized in this work but instead extended to multiprocessor scheduling of parallel DAG tasks.

Safari et al. [23] survey current techniques for fault tolerance in embedded systems from the point of view of power, energy, and thermal issues. They explain that some techniques for achieving fault tolerance come with additional timing overhead and that they increase the power consumption of the processor cores they are executed, which can lead to temperatures reaching above safe limits. They conclude that it is inevitable to jointly consider reliability, power and energy, timing, and temperature within embedded fault-tolerant systems. This is due to hardware-level countermeasures employed to contain chip temperatures within safe operating levels potentially hindering the ability to provide reliability and timing guarantees. They suggest applying multi-objective optimizations for future research relating to fault-tolerant embedded systems.

In recent work by Abeni et al. [24] they present a fault-tolerant cloud-native environment for real-time applications called *FTRTC*, which provides an adjustable level of fault tolerance. Unlike traditional time-critical applications, cloud-based applications often utilize distributed architectures for scalability and elasticity demands, at the cost of increased complexity. From streaming services and social media to workplace collaboration tools and edge computing, cloud-native applications are becoming intertwined in many parts of society. Cloud-native applications are applications designed for a cloud computing architecture. Cloud applications often consist of many microservices that can be deployed locally or distributed over multiple machines [24]. The concept of edge computing can be paired with cloud applications to improve the quality of service and service latency. Compared to a centralized cloud computing platform, edge computing moves the networking, control, computing, and storage closer to the edges of the cloud architecture [25]. There are prior works addressing fault tolerance in cloud-based environments such as that of Malik and Huet [26], however, they did not cover microservices or cloud-native applications as concepts. The FTRTC project leverages microservices to design a real-time fault-tolerant cloud computing platform. Abeni et al. underscores the importance of fault-tolerant real-time applications in the growing market of cloud

computing.

In a recent article by Reghenzani et al. [27], they survey state-of-the-art scientific work and explore future research questions for fault tolerance of hardware faults in real-time systems, primarily from the perspective of software-based solutions. They explore the concept of Software-Implemented Hardware Fault-Tolerance (SIHFT), explaining how SIHFT solutions ease production development and positively impact hardware costs. Additional custom hardware components are not required through software-based approaches, which is additionally important if Commercial Off-The-Shelf (COTS) hardware is to be employed in critical systems. Reghenzani et al. formulate many unanswered future research questions relating to fault tolerance. They state that the DAG task model presents many challenges and opportunities for future work, and specifically raise the question of how to efficiently schedule fault-tolerant algorithms when DAG task models are employed; a question this thesis aims to answer.

3
Problem Formulation

Pairing the parallel DAG task model with a fault tolerance model is a seemingly novel area within real-time systems and schedulability theory. Therefore, this thesis aims to examine scheduling approaches for parallel DAG tasks that can appropriately handle the additional execution time introduced due to fault tolerance mechanisms on a multiprocessor architecture. More specifically:

> Given a set of n different sporadic DAG tasks, a multiprocessor platform with m processors, and a fault model assuming f number of transient faults within D_{max}, how can the faults be tolerated such that all task deadlines are met? More specifically, we want to propose a test to determine the minimum number of required processors to schedule the n tasks under the assumed fault model while providing a reasonable accuracy in the result.

Inspired by the work of Melani et al. [3] the purpose of this thesis is to propose fault-tolerant schedulability analysis for a single DAG task, assuming a work-conserving scheduling algorithm. More specifically, given a collection of DAG tasks, this work will propose techniques that will determine the minimum number of processors that will be required for each DAG task, using proposed schedulability tests for single DAG tasks.

There has been extensive work done within the fields of parallel tasks on multiprocessors[2][3] and fault tolerance of sequential tasks on multiprocessors [28][16]. However, examining the combination of fault tolerance and parallel tasks on a multiprocessor architecture is important and has, to the best of our knowledge, not been addressed in this manner earlier. It is of interest to be able to provide assurances of parallel task execution, even in the presence of faults, i.e. be able to determine whether or not tasks meet their deadline given some taskset and fault model, on a multiprocessor architecture.

The prominence of multiprocessor architectures creates challenges for guaranteeing timely execution in critical systems. The work of this thesis is essential for developing fault-tolerant and efficient mechanisms for such systems using offline analysis. Providing correct execution of parallel tasks in the presence of faults under a certain fault model requires the development of appropriate scheduling algorithms and their analysis. To this end, temporal redundancy will be employed to achieve fault tolerance for the assumed system model, and a scheduling algorithm will be proposed.

The complexity of finding the workload of a DAG task in the worst case increases when anticipating faults, i.e. finding W^f_{max}, and fault recovery mechanisms increase

the system workload during runtime. Consequently, it is important to consider a fault model that accurately predicts the potential faults the system can suffer during runtime, to not over- or underestimate their potential impact.

3.1 Challenges

There are many challenging aspects to consider, and the following list aims to explicitly highlight the key challenges of the work:

- No fault-tolerant system can tolerate an infinite number of faults, and we therefore need to model the expected faults in such a way that they represent the world and expected environment. In this thesis, we consider transient faults.

- Additional hardware and software mechanisms can be utilized to detect faults, and there are different backup techniques, such as running backup tasks in parallel or re-executing the entire DAG task, that can be employed to handle fault recovery. This thesis addresses the challenge of modeling recovery mechanisms to minimize the additional required hardware.

- Finding the worst-case workload and the critical path of a DAG task in the presence of faults is not trivial in the general case. There are well-established tests to compute the response time of a DAG task without considering faults. However, in the presence of faults, the workload computation has to account for faults and the worst-case situation is therefore harder to find. A straightforward response time estimate equation for a DAG task is that of Melani et al. [3], which can be utilized to account for fault tolerance. This test uses the critical path and the worst-case workload of a DAG task. Therefore, a challenge is to extend this test to compute the workload and critical path in the presence of faults.

- Depending on the number of faults, the longest path of a DAG task can change from one instance to another, thus presenting a problem with exponential complexity in the general case. Therefore, a challenge is to compute the makespan of DAG tasks in polynomial time.

- A final challenge is reducing pessimism in the proposed tests, as excessive pessimism can lead to overly conservative estimates, introducing unnecessary overhead and limiting their practicality and usefulness.

3.2 Limitations

A few aspects are limiting the scope of the proposed work, which are the following:

- This thesis focuses mainly on constrained deadline tasks. There are other task models, such as those with arbitrary deadlines, which are not considered.

- This thesis only considers temporary faults in hardware and software. That is, this thesis does not consider the possibility of permanent faults, i.e. faults that occur on the program level like incorrectly initialized global variables or permanent processor failures.

4

Algorithms and Analysis

This chapter will present the fault-tolerant analysis and algorithms for examining the schedulability of single DAG tasks and multiple DAG tasks, where a total of six schedulability tests will be presented. It will contextualize previous analysis and explain how it is used to cover the fault-tolerant domain. Analyzing the fault-tolerant scheduling of a single DAG task is quite challenging, which is therefore addressed first. We will then examine how this can be extended to guarantee the schedulability for multiple tasks.

Section 4.1 will present the common scheduling algorithm assumed for the rest of the analysis. Section 4.2 will introduce the common theory for subsequent sections. Section 4.3 presents a trivial approach for examining all fault occurrences within a DAG task, and how it can be used to determine schedulability. The following sections present different methods for examining fault-tolerant scheduling of single and multiple tasks.

4.1 Scheduling Algorithm

The work of this thesis adopts a fault-tolerant work-conserving scheduling algorithm, assuming a non-preemptive setup in which once a node begins its execution it will continue uninterrupted until completion.

Let us first consider the fault-free case for a work-conserving scheduling algorithm. A work-conserving algorithm always attempts to keep its resources busy. If there is an idle processor and a task or node is ready to be scheduled, it will schedule that task or node to the idle processor. The ready nodes are kept in a ready queue. Work-conserving scheduling algorithms are commonly found in many prior works, including the work of Melani et al. [3], Jiang et al. [20], and He et al. [13]. A work-conserving scheduling algorithm is a general algorithm that is agnostic to the priority ordering of tasks, i.e. it can be used with fixed and dynamic priority tasks. The usage of a work-conserving scheduler in this work therefore allows the scope and analysis to apply to a wide range of task models. Algorithm 1 presents a work-conserving scheduling algorithm for a single task G.

Algorithm 1: A work-conserving scheduling algorithm for a DAG task

Input: Task G, number of processors m
Output:
1 RQ $\leftarrow v_{src}$
2 **while** RQ $\neq \emptyset$ **and** some processor is idle **do**
3 $s \leftarrow$ Extract at most m nodes from RQ
4 RQ $\leftarrow$ RQ $\setminus s$
5 Dispatch all nodes in s on the m processors
6 **if EVENT**: a node v completes its execution **then**
7 RQ $\leftarrow$ RQ $\cup$ successors of v

We will now propose an extension of Algorithm 1, which is augmented to include fault recovery. As mentioned, faults are assumed to be detected at the node level. Therefore, the faulty node will be re-executed as soon as a fault is detected. Importantly, in this thesis, we will not employ any fault-detection mechanism. We assume that faults can be detected and we presume a 100% error-detection coverage. However, for practical systems, if the detection coverage is less than total coverage, we can instead provide a probabilistic analysis for the lower detection rate to guarantee the reliability of our system. Probabilistic analysis will not be considered in this paper.

Algorithm 2 presents the work-conserving scheduling algorithm assumed for the subsequent analysis. The ready queue RQ is initialized to contain the source node of a task G. While RQ is not empty and at least one processor is idle, up to m nodes are extracted from RQ to be executed on the m processors. If an error is detected in a selected node v, the erroneous node will be instantly re-executed once more on the same processor. Instead, if no error is detected, then the successors of v are appended to RQ after v has finished execution.

Algorithm 2: The common work-conserving scheduling algorithm

Input: Task G, number of processors m
Output:
1 RQ $\leftarrow v_{src}$
2 **while** RQ $\neq \emptyset$ **and** some processor is idle **do**
3 $s \leftarrow$ Extract at most m nodes from RQ
4 RQ $\leftarrow$ RQ $\setminus s$
5 Dispatch all nodes in s on the m processors
6 **if EVENT**: an error is detected in a node v during its execution **then**
7 Execute v from the beginning on the same processor
8 **if EVENT**: a node v completes its execution **then**
9 RQ $\leftarrow$ RQ $\cup$ successors of v

This thesis deals with the scheduling of a single task and the scheduling of multiple tasks in the form of a taskset. Federated scheduling is used to schedule multiple tasks, where each processor employs the work-conserving scheduling algorithm of Algorithm 2.

4.2 Parallel DAG Task Analysis

Regardless of the complexity and size of a DAG task, the makespan analysis of Melani et al. [3] primarily utilizes only two static parameters of a task, abstracting it using the length of the longest path, L_i, and its total workload, W_i. Recall that the actual execution sequence is determined during runtime and is therefore unknown for offline analysis. As the number of nodes in a DAG task grows, so does the number of potential execution sequences derived from the DAG. There are many possible execution sequences from even a simple DAG structure. Therefore, it is beneficial to provide estimates and analysis using the static parameters of a DAG task, as they are not dependent on a specific execution sequence. Melani et al. [3] provide an equation for finding a response time estimate using only static task parameters, which is vital to the work of this thesis. The equation is shown in Equation 4.1,

$$R_i \leq L_i + \frac{W_i - L_i}{m} \tag{4.1}$$

For an arbitrary DAG, it is trivial that $L_i \leq W_i$, as W_i inherently includes the work of L_i, as well as the rest of the work of the DAG. Melani et al. [3] expresses the following relationship in the fault-free case between the length of the longest path L_i, the total workload W_i, and the length of the critical path L^*,

$$L^* \leq L_i \leq W_i \tag{4.2}$$

The critical path is not necessarily the longest path of a DAG task. Recall that the critical path is the path suffering the most interference. Figure 4.1 exemplifies an execution sequence of a DAG task where the critical path is not the longest path. The WCET is shown on the top half of each node. As v_1 and v_8 are shared for all paths, nodes 2 to 7 are the relevant ones in this scenario. The path through v_2 and v_5 make up the critical path for the specific execution sequence shown on the right. In contrast, the longest path is routed through v_3 and v_6, since node v_6 has a WCET of 2. The critical path suffers a total of 2 time units of interference. When referring to a single task, the task index of a node $v_{i,j}$ is omitted for clarity, unless mentioned explicitly.

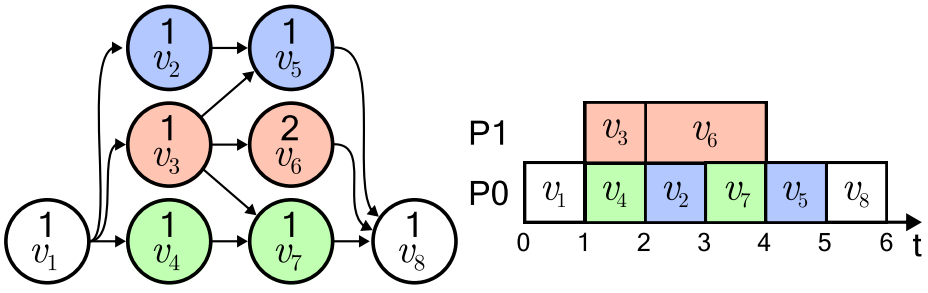


Figure 4.1: An example DAG and arbitrary execution sequence, highlighting how the critical path does not necessarily equal the longest path. The number on the top half of each node represents the WCET. The total workload $W = 9$

Another execution sequence could be constructed by allowing the v_2 to execute before v_4 at $t = 1$, forcing v_7 to finish last before the final node v_8, as shown in Figure 4.2. This shows that there are many possible execution sequences that one has to consider during the analysis of the algorithm. However, considering all such scenarios is not computationally feasible.

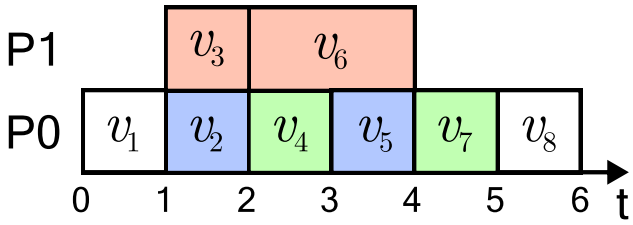


Figure 4.2: Alternative execution sequence to that of Figure 4.1, instead prioritizing v_2 over v_4 at $t = 1$

Ultimately, the subsequent analysis will explore the relationship between the total workload W_i, the length of the critical path L^*, and the longest path L_i. Equation 4.2 states that the length critical path λ^* is bounded by the length of the longest path L_i, which can therefore be used instead of the critical path. As a result, the estimate will either be larger or equal to that of using the critical path, but can be determined before running the system, i.e. offline using only static parameters of the DAG. When the critical path is not the longest, it would be pessimistic to assume it equals the longest path.

In the analysis of DAG tasks without considering faults the length of the longest paths, and parameters like the workload, will be static. However, in the fault-tolerant domain, the longest path in the fault-free case may become shorter than another path when considering faults, as the additional execution time required for fault recovery could be different for the two paths.

Figure 4.3 highlights the change of the longest path under faults. The two paths of the DAG are $\lambda_1 = \{v_1, v_2, v_4, v_5\}$ and $\lambda_2 = \{v_1, v_3, v_5\}$. The DAG in the fault-free case is shown on the left. Two faults are assumed to afflict v_3. The additional recovery time introduced by faults has been included and embedded in the execution time of the node. Such recovery needs to be modeled and is therefore represented by $c(v_{i,j}, f) = c_{i,j} \cdot (f + 1)$. In the fault-free case, λ_1 is the longest path, where $len(\lambda_1) = c_1 + c_2 + c_4 + c_5 = 1 + 2 + 2 + 1 = 6$, whereas $len(\lambda_2) = c_1 + c_3 + c_5 = 5$. However, as shown on the right of Figure 4.3 the number of faults affecting each path determines the longest path. Examining the two paths shows that λ_2 is longer than λ_1 in the faulty case. Let $len(\lambda, f)$ denote the length of a path λ under f faults, then $len(\lambda_1, 2) = 10$, $len(\lambda_2, 2) = 11$. Assuming a single fault affects v_3 and v_4 respectively, then $len(\lambda_1, 1) = len(\lambda_2, 1) = 8$, further highlighting how the longest path is dependent on the number of faults suffered by each path.

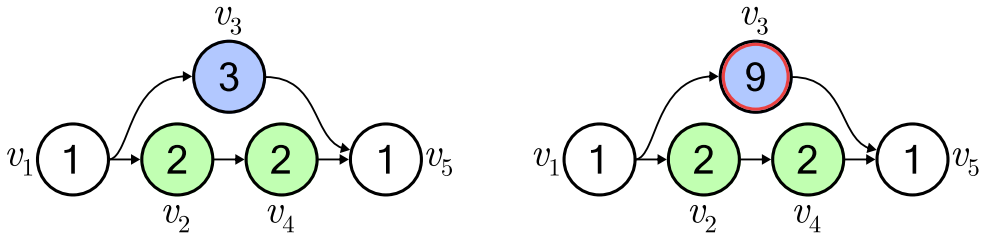


Figure 4.3: Left: Example DAG assuming no faults; Right: Same DAG assuming the two faults occur in v_3, highlighting how the longest path changes when considering faults

Since faults are to be considered throughout the upcoming analysis, the notion of a *Fault-Recovery DAG* (FRDAG) is introduced for clarity. The example DAG on the right of Figure 4.3 is a FRDAG where $f = 2$. By embedding the fault recovery time into the execution time of the affected nodes, an FRDAG task can be viewed and examined as a separate DAG task. If nothing else is stated, when faults are considered the terms DAG and task refer to a FRDAG.

The way Equation 4.1 of Melani et al. [3] works is by maximizing the critical path and maximizing the workload. The node which maximizes the workload might not be the same node which maximizes the length of the longest path. In this thesis, three different approaches will be considered: 1. Separate maximization of L and W, 2. Joint maximization of L and W, 3. Extending the separate approach to utilize more than one long path of a DAG task. Maximizing L and W requires us to consider how faults can affect the nodes of the DAG. To that end, we will consider different faulty scenarios in the next subsection.

4.3 Exhaustive Fault Scenario Search

Examining all possibilities of f faults occurring in a DAG task G, and finding which scenario produces the maximum makespan of G is a safe but computationally expensive approach.

The exhaustive case will cover all the scenarios under which the f can occur in many different nodes. Among all of the scenarios, one will cause the makespan to be maximized, i.e. the worst-case fault occurrence. Therefore, it will find the upper bound of the response time of G under f faults. The worst-case scenario occurs under three conditions:

1. All f faults occur
2. Faults are detected at the end of the execution of nodes
3. The same node can suffer multiple faults

Firstly, the workload of a system will only increase when considering a larger number of faults, thus assuming all f faults occur will be worse in terms of workload than if less than f faults are assumed. Secondly, if faults were to be detected earlier

than at the end of execution, then the additional recovery time, and therefore also L_i and W_i, would decrease. Thirdly, the node with the maximum WCET of a task is the node that would produce the largest recovery time if afflicted by all f faults.

To exemplify the meaning of a fault scenario in a DAG task G, assume that G is made up of two nodes and that two faults can occur. Let (a, b) represent a fault scenario, where a and b are the number of faults suffered by each respective node. There are a total of six ways the faults can affect the nodes: $(0, 0)$,$(0, 1)$,$(1, 0)$,$(1, 1)$,$(2, 0)$, $(0, 2)$. However, as we want to find the WCFO, we need to consider the cases when all potential faults occur. Therefore, only scenarios considering two faults will be examined, which leaves: $(1, 1)$, $(2, 0)$, $(0, 2)$.

To find the number of possible fault scenarios of a DAG task G, let n denote the number of nodes in G, i.e. $n = |V|$, and f the number of faults affecting the nodes in G. Equation 4.3 computes the number of fault scenarios that can occur in the nodes in V, where all f faults occur in each scenario. Notably, the ordering of the faults is inconsequential. The problem can be viewed as a combinatorial *stars and bars*-problem [29], where it is reduced to placing indistinguishable balls into distinguishable bins. The results of Equation 4.3 also represent the number of FRDAGs derived from the single original task.

$$\binom{n+f-1}{f} = \frac{(n+f-1)!}{(f!((n+f-1)-f)!} = \frac{(n+f-1)!}{f!(n-1)!} \tag{4.3}$$

Each possible fault scenario in a DAG task assumes a certain distribution of the f faults across all nodes of the task, where each node can suffer 0 to f faults. Examining all fault scenarios is computationally expensive because, from even a single DAG, there could be many fault scenarios to examine to find the one that maximizes the response time. By embedding the fault recovery time of each fault into the WCET for the affected nodes for each fault scenario using a FRDAG, a new collection of DAG tasks is created from a single fault-free task. Importantly, the number of possible fault scenarios is dependent on the number of faults and the number of nodes in G.

Each FRDAG can be viewed as a separate DAG. Using the response time analysis of Melani et al. [3] for each FRDAG will result in a set of response time estimates. The response time in the worst-case, i.e. the largest response time in the set, will correspond to the fault scenario maximizing the additional computational workload introduced by the f faults. To provide a sufficient test, the worst-case scenario has to be included and accounted for. Therefore, by examining the scenario producing the largest response time, the schedulability of the original task under f faults is covered by exhaustively examining all fault scenarios.

Exhaustively examining every fault scenario is computationally expensive, and is therefore less desirable in practice. However, in this paper, we will propose a polynomial-time algorithm that covers the worst-case fault occurrence.

4.4 Bounding Response Times using Separate Maximization of L and W

This section presents a schedulability test based on the single DAG response time test of Melani et al.[3] shown in Equation 4.1, which has the benefit of only using the total workload, W_i and a single longest path, L_i. Separate maximization of W_i and L_i means that we separately calculate the effects of faults on W_i and L_i. In other words, we assume all f faults occur in the node which maximizes W_i, and we find the longest path assuming each path suffers f faults. Using separate maximization and inputting the maximized values of W_i and L_i in Equation 4.1 provides a safe estimate for the response time of a DAG task. It is safe since it will produce an overestimated response time. However, if the sum of the two addends of Equation 4.1, L_i and $\frac{W_i - L_i}{m}$, can be maximized at the same time, it will produce a tighter response time of a task, than if they are maximized separately. Such maximization is examined in Section 4.5.

How faults occur in a DAG task will affect whether or not L_i and W_i are maximized, and finding this occurrence can be quite difficult if we want to maximize both L_i and W_i. For some fault scenarios, W_i is maximized, whereas for others L_i is maximized. In terms of total work done, the node with the largest WCET in the task will be the node resulting in the largest recovery time if afflicted by all f faults. Therefore, assuming all f can occur anywhere in DAG, the node with the largest WCET is the node that will maximize W_i if afflicted by f faults. Similarly, individual paths are maximized when all f faults affect the node in each path with the largest WCET.

Assuming W_i and L_i can be calculated in the fault-free case, a safe approach is to maximize L_i independently, and W_i independently, each using all f faults. We call this approach *separate maximization*. Separate maximization under f transient faults will provide an upper-bound estimate for the response time for a single DAG task, where two cases need to be considered:

1. All faults affect the critical path

2. All faults occur in any node of the DAG so that the total work is maximized

Lemma 1. *Consider a path λ of a task G. Let c_{max} be the maximum worst-case execution time of any node of λ. The workload of λ, assuming f faults occur in the nodes of λ during the interval between the release of an instance of G and its deadline D, is bounded by the fault-free workload of λ plus the additional workload of all faults occurring in the node that has WCET of c_{max}.*

Proof. For a sequence of instances of G, assume an arbitrary instance of G where at most f faults occur within the interval between its release and deadline D. For a path $\lambda = \{v_1, v_2, ..., v_x\}$ of G, assume a scenario where at most f faults affect the nodes in λ, such that for $i \in [1, x] : v_i$ suffers f_i faults, where $\sum_{i=1}^{x} f_i \leq f$. Let c_{max} be the maximum worst-case execution time of any node of λ, such that $c_{max} = \max_{v_i \in \lambda}\{c_i\}$. For each node v_i in λ, it is assumed that the additional workload introduced due to

faults is $f_i c_i$, as fault recovery is done by re-execution. Note that it is possible for $f_i = 0$ for some i. The fault-free workload of λ is $len(\lambda)$. The resulting workload of a node suffering f_i faults is its fault-free workload plus the additional workload due to fault recovery, i.e. $c_i^f = c_i + f_i c_i$. The workload of λ under f faults can be expressed as $\sum_{i=1}^{x}(c_i + f_i c_i)$. As $\forall i : c_i \leq c_{max}$, it implies $f_i c_i \leq f_i c_{max}$, meaning the additional workload introduced due to faults for any node $v_i \in \lambda$, $f_i c_i$, is bounded by $f_i c_{max}$. Therefore, $c_i + f_i c_i \leq c_i + f_i c_{max} \Rightarrow \sum_{i=1}^{x}(c_i + f_i c_i) \leq \sum_{i=1}^{x}(c_i + f_i c_{max})$. Similarly, as $\sum_{i=1}^{x} f_i \leq f$, it implies $\sum_{i=1}^{x}(c_i + f_i c_{max}) \leq \sum_{i=1}^{x} c_i + f c_{max}$. Therefore,

$$len(\lambda) + \sum_{i=1}^{x} f_i c_i \leq len(\lambda) + f c_{max} \tag{4.4}$$

□

As shown in Section 4.2, the longest path of a task in the fault-free case can become shorter than another path in the faulty case. This is due to the additional workload of fault recovery extending the length of the paths by different amounts, as the maximum WCET of the nodes in each path might differ. Therefore, creating an algorithm for finding the new longest path when assuming a certain number of faults is required. This implies that we first need to find all possible complete paths of a DAG task $G = (V, E)$, to know which is the largest under f faults. The common Depth-First Search (DFS) algorithm [30] can be employed using the nodes of V.

Lemma 1 points out that the workload of a path λ in a task G suffering from faults can be bounded using the maximum WCET of the nodes in λ. There are typically multiple paths in a DAG, and therefore the maximum work generated by any path in task G affected by f faults is denoted L_{max}^f.

The workload of any path λ suffering f faults is upper-bounded by L_{max}^f. Finding the effect of faults on the entire workload of a task is also important, as the workload is a vital static parameter in schedulability test proposed by Melani et al. [3]. W_{max}^f denotes the maximum workload generated in a DAG task G suffering from f faults. Notably, any node of a task can contribute to the computation of W_{max}^f.

Lemma 2. *In the presence of f faults during the interval between the release of an instance of a DAG task $G = (V, E, D)$ and its deadline D, the total workload of G cannot be larger than the maximum workload under faults W_{max}^f.*

Proof. For a sequence of instances of G, assume an arbitrary instance of G where at most f faults occur within the interval between its release and deadline D. For a set of nodes $V = \{v_1, v_2, ..., v_n\}$ of $G = (V, E)$, let W be the workload of G in a fault-free scenario. Assume a scenario where at most f faults affect the nodes in V, such that for $i \in [1, n] : v_i$ suffers f_i faults, $\sum_{i=1}^{n} f_i \leq f$, and $n = |V|$. Let c_{max} be the maximum worst-case execution time of any node in V, such that $c_{max} = \max_{v_i \in V}\{c_i\}$. For each node v_i in V it is assumed that the additional workload introduced due to faults is $f_i c_i$, as fault recovery is done by re-execution. Note that it is possible for $f_i = 0$ for some i. The resulting workload of a node suffering f_i faults is its fault-free workload plus the additional workload due to fault recovery, i.e. $c_i^f = c_i + f_i c_i$. The workload of

the whole DAG task can be expressed as $\sum_{i=1}^{n}(c_i + f_i c_i)$. Assume $W^f_{max} = W + f c_{max}$. Similarly to Lemma 1, as $\forall i : c_i \leq c_{max}$, it implies $f_i c_i \leq f_i c_{max}$, meaning the additional workload introduced due to faults for any node $v_i \in V$, $f_i c_i$, is bounded by $f_i c_{max}$. Therefore, $c_i + f_i c_i \leq c_i + f_i c_{max} \Rightarrow \sum_{i=1}^{n}(c_i + f_i c_i) \leq \sum_{i=1}^{n}(c_i + f_i c_{max})$. As $\sum_{i=1}^{n} f_i \leq f$, it implies $\sum_{i=1}^{n}(c_i + f_i c_{max}) \leq \sum_{i=1}^{n} c_i + f c_{max}$. Recall that $W = \sum_{i=1}^{n} c_i$. Therefore,

$$W + \sum_{i=1}^{n} f_i c_i \leq W + f c_{max} \tag{4.5}$$

□

Before presenting the schedulability test using L^f_{max} and W^f_{max}, we will present pseudocode to show how the required static parameters can be computed in polynomial time for a given DAG. Algorithm 3 showcases how the source node, sink node, c_{max}, and W of a DAG task G are found. One pass over all nodes in V is required, and therefore its runtime complexity is $O(|V|)$. This algorithm is required in the other algorithms proposed in this work. By iterating over each node, its WCET is added to a sum W, and compared to a current maximum WCET c_{max}. If the node has no predecessors, then it is a source node. Likewise, if the node has no successors, it is a sink node. The source node, the sink node, c_{max}, and W are found once the entire DAG has been traversed.

Algorithm 3: Finding $v_{source}, v_{sink}, c_{max}, W$ of a task G

Input: $G = (V, E)$
Output: $v_{source}, v_{sink}, c_{max}, W$

```
1  v_source ← null
2  v_sink ← null
3  W ← 0
4  c_max ← 0
5  for i ← 1 to |V| do
6      W ← W + c_i
7      if c_i > c_max then
8          c_max ← c_i
9      if predecessors(v_i) == ∅ then
10         v_source ← v
11     if successors(v_i) == ∅ then
12         v_sink ← v
13 return v_source, v_sink, c_max, W
```

We will now provide Algorithm 4, which is based on DFS to find the list of paths P, list of path lengths L, list of maximum WCETs found in each path C^L_{max}, L^f_{max}, and W^f_{max} of a task G. As these parameters are found by traversing the DAG from the source node to the sink node, the runtime complexity follows that of DFS, i.e. $O(|V| + |E|)$. The usage of Algorithm 3 does not impact the time complexity, as $O(2|V| + |E|) = O(|V| + |E|)$. The algorithm works by following the edges of a node from the source node to the sink node. The length of the path is continuously

updated, as is the currently largest WCET found in the path c_{max}^{path}. Once the sink node has been reached, a path has been completed. Therefore, the path is added to P, c_{max}^{path} is added to C_{max}^{L}, and the length of the path is added to L. Using the path length L, the faulty path length L^f is calculated and compared to the currently largest faulty path L_{max}^{f}. The final value of L_{max}^{f} equals the longest faulty path of the task. The function $\texttt{DFS*}_{\texttt{rec}}$ is recursively called for each successor of a node, following the depth-first approach.

Algorithm 4: `DFS*`

Input: $G = (V, E), f$
Output: $P, L, C_{max}^{L}, L_{max}^{f}, W_{max}^{f}$

Function `DFS*`(V, f)**:**
- $v_{source}, v_{sink}, c_{max}, W \leftarrow$ Algorithm 3(V)
- $W_{max}^{f} \leftarrow W + c_{max} \cdot f$
- $L_{max}^{f} \leftarrow 0$
- $L \leftarrow []$ `// list of the lengths of all paths`
- $C_{max}^{L} \leftarrow []$ `// list of the maximum WCET of all paths`
- $P \leftarrow []$ `// list of all paths`
- $path \leftarrow [v_{source}]$
- $len \leftarrow 0$
- $P, L, C_{max}^{L}, L_{max}^{f} \leftarrow \texttt{DFS*}_{\texttt{rec}}(path, v_{sink}, len, L, L_{max}^{f}, c_{source}, C_{max}^{L}, P, f)$
- **return** $P, L, C_{max}^{L}, W, c_{max}, L_{max}^{f}, W_{max}^{f}$

Function $\texttt{DFS*}_{\texttt{rec}}(path, v_{sink}, len, L, L_{max}^{f}, c_{max}^{path}, C_{max}^{L}, P, f)$**:**
- $v_{end} \leftarrow path(end)$
- $len \leftarrow len + c_{end}$
- **if** $c_{end} > c_{max}^{path}$ **then**
  - $c_{max}^{path} \leftarrow c_{end}$
- **if** $v_{end} == v_{sink}$ **then**
  - $P \leftarrow P + [path]$
  - $C_{max}^{L} \leftarrow C_{max}^{L} + [c_{max}^{path}]$
  - $L \leftarrow L + [len]$
  - $L^f \leftarrow len + c_{max}^{path} \cdot f$
  - **if** $L^f > L_{max}^{f}$ **then**
    - $L_{max}^{f} \leftarrow L^f$
  - **return** $P, L, C_{max}^{L}, L_{max}^{f}$
- **for** $v \leftarrow successors(v_{end})$ **do**
  - $next \leftarrow path + [v]$
  - $P, L, C_{max}^{L}, L_{max}^{f} \leftarrow \texttt{DFS*}_{\texttt{rec}}(next, v_{sink}, len, L, L_{max}^{f}, c_{max}^{path}, C_{max}^{L}, P, f)$
- **return** $P, L, C_{max}^{L}, L_{max}^{f}$

From Lemma 1 and 2 together with the notion of L_{max}^{f}, it is known that for a task G under f faults, we can bound its longest path by L_{max}^{f} and its total workload by W_{max}^{f}. Inspired by the work of Melani et al. [3], we propose a simple polynomial-time equation that can find an estimate of the makespan of a DAG task using its fault-aware longest path L_{max}^{f} and fault-aware worst-case workload W_{max}^{f}.

Theorem 1. *In the presence of f faults during the interval between the release of an instance of a DAG task $G = (V, E, D)$ and its deadline D, a response time estimate R of the instance is found by:*

$$R \leq L^f_{max} + \frac{W^f_{max} - L^f_{max}}{m} \tag{4.6}$$

Proof. For a sequence of task instances $G = (V, E, D)$, assume an arbitrary instance of G. In the worst case, the instance can at most suffer from f faults within the interval between the release time of the instance and its deadline D. For $V = \{v_1, v_2, ..., v_n\}$, assume an execution sequence in which at most f transient faults affect the nodes in V such that for $i \in [1, n] : v_i$ suffers f_i faults, $\sum_{i=1}^{n} f_i \leq f$, and $n = |V|$. For this sequence, assume the total amount of work done by all nodes of G in the presence of f faults is W^f_{arb}. It is known from Lemma 2 that the total workload of G cannot be larger than W^f_{max} in the presence of at most f faults, which implies that $W^f_{arb} \leq W^f_{max}$. For this execution sequence, there exists a path that finishes its execution last, i.e. the critical path. Let L^f_{arb} denote the length of the critical path in the presence of faults. We know the critical path must be bounded by L^f_{max}, implying $L^f_{arb} \leq L^f_{max}$. Let c_{max} be the maximum worst-case execution time of any node in V, such that $c_{max} = \max_{v_i \in V}\{c_i\}$. By assuming the worst-case scenario for the workload and the longest path, i.e. W^f_{max} and L^f_{max}, we can use Equation 4.1 to find the response time under this arbitrary execution sequence,

$$L^f_{arb} + \frac{W^f_{arb} - L^f_{arb}}{m} = L^f_{arb}(1 - \frac{1}{m}) + \frac{W^f_{arb}}{m}$$

Since $L^f_{arb} \leq L^f_{max}$,

$$L^f_{arb}(1 - \frac{1}{m}) + \frac{W^f_{arb}}{m} \leq L^f_{max}(1 - \frac{1}{m}) + \frac{W^f_{arb}}{m}$$

Since $W^f_{arb} \leq W^f_{max}$,

$$L^f_{max}(1 - \frac{1}{m}) + \frac{W^f_{arb}}{m} \leq L^f_{max}(1 - \frac{1}{m}) + \frac{W^f_{max}}{m}$$

Since L^f_{max} is an upper bound for L^f_{arb} and W^f_{max} is an upper bound for W^f_{arb}, the following equation produces a response time estimate for any sequence of G under f faults,

$$L^f_{max} + \frac{W^f_{max} - L^f_{max}}{m}$$

□

It should be noted that Theorem 1 provides a pessimistic response time estimate due to the following reason. If the node with the maximum WCET of a DAG, c_{max}, lies within the critical path producing L^f_{max}, then Theorem 1 could produce a larger estimate than if it did not. This is because the worst-case number of faults are

counted twice: first in estimating W^f_{max} and second in estimating L^f_{max}. Regardless of the location of c_{max}, the resulting W^f_{max} will remain the same. When the node with WCET c_{max} is outside the critical path, the computation for L^f_{max} and W^f_{max} assumes the f faults affect different nodes.

Figure 4.4 exemplifies two DAG task structures to highlight the effect of the position of c_{max}, where the execution times are shown in each node. Even though nodes v_2 and v_4 can be combined and viewed as a single sequential workload, these DAG structures are utilized for illustrative purposes. Assume a single fault and 2 available processors. The left DAG structure shows how node v_3 has a workload of $c_{max} = 4$, where it results in $W^f_{max} = W + fc_{max} = 12 + 4 \cdot 1 = 16$, and $L^f_{max} = 11$. However, swapping the execution times of node v_3 and v_4 instead produces $W^f_{max} = 16$ and $L^f_{max} = 13$. Depending on the location of c_{max}, the resulting L^f_{max} differs. Using Equation 4.6 of Theorem 1 results in the following response time estimates: $R_1 = 11 + \frac{16-11}{2} = 13.5$ and $R_2 = 13 + \frac{16-13}{2} = 14.5$.

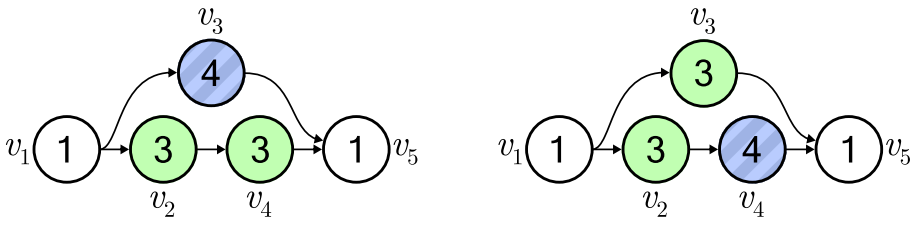


Figure 4.4: Left: Example DAG where c_{max} is in node v_3; Right: Example DAG where the execution times of node v_3 and v_4 of the left DAG has been swapped

Given a makespan estimate produced by Theorem 1 for a task G, the task meets its deadline D if the estimate is less than or equal to D.

Corollary 1. *In the presence of f faults during the interval between the release of an instance of a DAG task $G = (V, E, D)$ and its deadline D, all instances of G will meet their corresponding deadline if the response time estimate R of G from Theorem 1 is bounded by D, i.e. $R \leq D$.*

Algorithm 5 presents pseudocode for determining the schedulability of a task G in the presence of f faults, based on Corollary 1. It is the first of the six schedulability tests proposed in this work. The algorithm is denoted SDT (Single DAG Trivial).

Algorithm 5: SDT

Input: $G = (V, E, D)$, number of faults f, number of available processors m
Output: The sufficient schedulability of a G under f faults

```
1 _, _, _, _, _, L^f_max, W^f_max ← Algorithm 4(G, f)
2 R^f ← L^f_max + (W^f_max − L^f_max) / m
3 if R^f ≤ D then
4     G is schedulable with SDT
5 else
6     G is unschedulable with SDT
```

If it is assumed that L^f_{max} and W^f_{max} can be pre-computed, the time complexity of SDT is constant or $O(1)$. Conversely, if they cannot be pre-computed, the time complexity of SDT instead follows that of Algorithm 4 and DFS, i.e. $O(|V| + |E|)$.

4.4.1 Extension for the scheduling of multiple DAG tasks

Similar to SDT, the MDT (Multiple DAG Trivial) test proposed in this section uses separate maximization of W_i and L_i of a task. MDT extends the analysis of SDT to schedule multiple tasks under the federated scheduling paradigm. Federated scheduling divides the tasks of a system into heavy and light tasks. A set of dedicated processors is allocated for each heavy task, in an attempt to grant them enough resources to meet their deadlines. The remaining processors are shared between the light tasks, where they execute sequentially and follow partitioned scheduling.

To reiterate, this and the following sections regarding the scheduling of multiple DAG tasks, we introduce subscript notation to refer to specific tasks or parameters of tasks, e.g. task i is referred to as G_i. In federated scheduling, each task $G_i \in \Gamma$ requires a certain number of processors m_i. In the fault-free case, the number of processors required for a task G_i can be estimated by rearranging the response time bound in Equation 4.1 of Melani et al., which is shown in Equation 4.7.

$$L + \frac{W - L}{m} \leq D \qquad m \geq \left\lceil \frac{W - L}{D - L} \right\rceil \tag{4.7}$$

Since faults are to be considered, any fault-free algorithm for finding the required number of processors cannot be used, as it might not sufficiently allocate enough processors if a task were to be afflicted by faults. Therefore, to find the required number of processors in the worst case under federated scheduling for a task, in the presence of f faults, the condition of Corollary 1 is utilized to solve for the number of processors m, which is shown in Equation 4.8.

$$m \geq \left\lceil \frac{W^f_{max} - L^f_{max}}{D - L^f_{max}} \right\rceil \tag{4.8}$$

By using Equation 4.8 for each task G_i in a taskset Γ, a safe estimate for the minimum number of required processors M under f faults will be found. Additionally, a necessary condition is that $D > L^f_{max}$, as the result could become undefined or negative otherwise. Let N be the number of tasks in Γ, m_i the required number of processors for a task G_i, then $M = \sum_{i=1}^{N} m_i$. The pessimism of Equation 4.6 discussed in Section 4.4 is similarly present in Equation 4.8, which in short is due to both equations utilizing L^f_{max} and W^f_{max} at the same time. The effect of the pessimism is a potential overestimate of the minimum number of required processors under f faults.

In the fault-tolerant domain of scheduling a taskset Γ, understanding how faults can occur across Γ is important. Identifying the most critical case that may lead to maximized response times, and therefore also potentially missed deadlines, needs to be examined to determine whether Γ is schedulable. The fault scenarios occurring across Γ within D_{max} can generally be divided into two categories, to which the first category is to be examined in this section.

1. All faults occur in a single DAG task

2. The faults are spread across the DAG tasks in the taskset

For each DAG task $G_i = (V_i, E_i)$, the WCRT will be achieved when all f faults occur in the nodes of V_i. Therefore, by examining each task individually using the response time bound of Equation 4.6 of Theorem 1, and checking whether all tasks meet their deadlines, will provide a sufficient schedulability test. If the condition is true, then the taskset is schedulable. If the condition is false, the taskset might still be schedulable using other analyses but failed with this one.

Algorithm 6 presents pseudocode for determining the sufficient schedulability of a taskset Γ in the presence of f faults, based on Equation 4.8. It is the second schedulability test proposed in this work. For each task $G_i \in \Gamma$, we use Algorithm 4 to find only the relevant parameters L^f_{max} and W^f_{max}. The required number of processors m of a single task G_i is computed using its deadline D_i. The required number of processors M is continuously compared to the available number of processors m. If M is greater than the number of available processors m, then the taskset is not schedulable using MDT. The test succeeds if the loop completes, i.e. if the total for all tasks is less than m.

Algorithm 6: MDT

Input: Taskset Γ, number of faults f, number of available processors m
Output: The sufficient schedulability of Γ under f faults

```
1 M ← 0
2 for G_i = (V_i, E_i, D_i) ← G_i ∈ Γ do
3     _, _, _, _, _, L^f_max, W^f_max ← Algorithm 4(G_i, f)
4     m_i ← ⌈(W^f_max − L^f_max) / (D_i − L^f_max)⌉
5     M ← M + m_i
6     if M > m then
7         return; Γ is unschedulable with MDT
8 Γ is schedulable with MDT
```

Similarly to SDT, the time complexity of MDT depends on whether it is assumed that L^f_{max} and W^f_{max} can be pre-computed. Let n denote the number of tasks in Γ. Assuming L^f_{max} and W^f_{max} can be pre-computed, as the required processors are computed for each task the time complexity of MDT is $O(n)$, and $O(n(max(|V_i| + |E_i|))$ if not.

4.5 Bounding Response Times using Joint Maximization of L and W

The SDT algorithm estimates a single DAG task's response time as the sum of its critical path length and the interference from non-critical path nodes. The separate maximization approach of Section 4.4 assumes that the total number of anticipated faults occur in both the critical path and in another node not in the critical path

at the same time. This is pessimistic, as the worst-case scenario of faults occurring in the DAG is accounted for in two ways—the scenario of faults occurring in the DAG which produces L^f_{max} and the scenario which produces W^f_{max}. Conversely, by examining each complete path of the system and attempting to maximize the entirety of Equation 4.1 under f faults, the pessimism of Theorem 1 can be reduced and a response time estimate closer to the actual response time could be achieved. When the node with the maximum WCET of a DAG task does not lie in the longest faulty path under the maximum assumed number of faults, a potentially smaller estimate than SDT can be found. A key insight is that the resulting total workload when examining paths separately is then becoming less than that of W^f_{max}.

An alternate approach to that of Section 4.4 and separate maximization, is therefore to jointly compute the effects of faults on the total workload and length of the longest path for a given number of faults. First, it is crucial to recognize that the longest path of a DAG task depends on the number of faults that task suffers, i.e., it can change as the number of faults varies. The term L^f_{max} of a DAG task is the length of the longest path when afflicted by all faults considered in the fault model. Therefore, L^f_{max} is dependent on the value of f, which is exemplified in Figure 4.3.

We are interested in finding the worst-case occurrences of the f faults in the system, i.e. the WCFO. Equation 4.1 considers only two main terms: the length of the longest path L_i and the workload without the longest path $\frac{W_i - L_i}{m}$. To find the WCFO, we aim to maximize both the longest path and the remaining workload.

Instead of independently maximizing the length of the longest path (L^f_{max}) and the total workload of the system (W^f_{max}), the joint analysis of this section assumes that each path of a task G is examined separately and suffers q number of faults. Each value of $q \in [0, f]$ is examined for each path, where the remaining $f - q$ faults affect the remaining nodes not in the currently examined path.

For a given path λ of a task G, assume the maximum length of λ can be computed for all possible values of q starting from 0 to f, and is denoted L^q. Similarly, assume the total workload of G can be computed where the rest of the nodes not in λ is assumed to suffer $f-q$ number of faults. This workload is denoted W^f. Importantly, W^f includes the recovery time for all f faults, regardless of whether they occur in λ. Equation 4.1 can be modified to use instead L^q and W^f. Taking the maximum out of the response time estimates found when examining q from 0 to f, results in a worst-case estimate for G when assuming the q faults occur in λ, which is shown in Equation 4.9.

$$\max_{q=0}^{f} \left\{ L^q + \frac{W^f - L^q}{m} \right\} \tag{4.9}$$

Finding the maximum possible makespan of a DAG task under a certain number of faults is desired since it is the worst-case scenario. A DAG task can have many paths, and the fault pattern in Equation 4.9 needs to be considered for all possible paths in a task to find the worst case. Let p denote the number of paths in a task, then Equation 4.9 needs to be computed p times. One of the paths will produce the maximum response estimate for some value of q and will be a safe worst-case estimate of the response time of G under f faults. This estimate can potentially offer a tighter estimate than Algorithm 5. The subscript j is introduced to enumerate the paths.

Theorem 2. *Let p denote the number of paths in a DAG task $G = (V, E, D)$, where the paths are numbered from 1 to p. Let L_j^q denote the maximum length of path j suffering q faults, where the remaining nodes not in the path suffer $f - q$ faults. Let W_j^f denote the maximum total workload of G accounting for the recovery of all f faults, including the q faults accounted for in L_j^q. Then, a response time estimate R of G in the presence of f faults within an interval of D can be expressed as:*

$$R \leq \max_{j=1}^{p} \left\{ \max_{q=0}^{f} \left\{ L_j^q + \frac{W_j^f - L_j^q}{m} \right\} \right\}$$

Proof. For a sequence of instances of G, assume an arbitrary instance of G where at most f faults can occur within the interval between its release and its deadline D. Assume a particular instance of G where x faults occur, such that $x \leq f$. One of the paths in G will be the longest when accounting for x faults through re-execution. Assume this longest path suffers q faults and the remaining nodes of G not in this path suffers l faults such that $x = l + q \leq f$. Under such a fault scenario, G can be viewed as an FRDAG task by embedding the fault recovery time in afflicted nodes, which enables the use of Equation 4.1. Let W^x represent the total workload including the additional work introduced by the x faults, and L^q the length of the longest path suffering q faults. It can then be expressed as:

$$L^q + \frac{W^x - L^q}{m}$$

When considering the worst case, as the number of faults in the system increases, the workload and the longest path cannot decrease. Therefore, we only need to consider $x = f$, as this is the worst-case. However, since the number of faults per path and the exact instance of G is not known, we need to consider all paths, due to the number of faults dictating the longest path in G. If the path suffering q faults is not the longest, the remaining nodes suffering $f - q = l$ faults will instead produce the longest path. Therefore, the worst case scenario is included by examining each path separately, and examining all possible number of faults that can afflict each path and finding whichever produces the largest response time estimate. Let L_j^q be the maximum length of path j when suffering q faults, and W_j^f be the total workload of G under all f faults, considering the q faults in path j and l faults in the remaining nodes. The response time of G can then be expressed as:

$$R \leq \max_{j=1}^{p} \left\{ \max_{q=0}^{f} \left\{ L_j^q + \frac{W_j^f - L_j^q}{m} \right\} \right\}$$

□

Algorithm 7 presents pseudocode for determining the sufficient schedulability of a task G in the presence of f faults, based on Theorem 2. The algorithm is denoted SDJ (Single DAG Joint). In essence, SDJ offers the same or better performance than SDT, since W_j^f is calculated dynamically depending on the values of q and which path is examined.

Algorithm 7: SDJ

Input: $G = (V, E, D)$, number of faults f, number of available processors m
Output: The sufficient schedulability of G under f faults

$P, L, C^L_{max}, W, c_{max}, \_, \_ \leftarrow$ Algorithm 4(G, f)
$R_{max} \leftarrow 0$
for $j \leftarrow 1$ *to* $|P|$ **do**
    $len \leftarrow L[j]$
    $c^L_{max} \leftarrow C^L_{max}[j]$ `// maximum WCET in path` j
    $c^W_{max} \leftarrow \max_{v \in (V \setminus P[j])} \{c\}$ `// maximum WCET outside of path` j
    for $q \leftarrow 0$ *to* f **do**
        $W(f, q) \leftarrow W + (f - q) \cdot c^W_{max} + q \cdot c^L_{max}$
        $L(q) \leftarrow len + q \cdot c^L_{max}$
        $R \leftarrow L(q) + \dfrac{W(f, q) - L(q)}{m}$
        if $R > R_{max}$ **then**
            $R_{max} \leftarrow R$
            if $R_{max} > D$ **then**
                G is unschedulable with SDJ
G is schedulable with SDJ

All output parameters of Algorithm 4 are assumed to be pre-computed, and the largest WCET of any node not in the currently examined path, c^W_{max}, can also be pre-computed for each path. Therefore, the time complexity of SDJ is $O(|P|f)$ as both the number of paths and faults are looped over.

4.5.1 Extension for scheduling of multiple DAG tasks

Extending the single DAG analysis of SDJ to consider the scheduling of multiple DAG tasks will be similar to the extension between SDT to MDT. Assuming the federated scheduling paradigm, the set of tasks is denoted Γ.

To schedule tasksets, finding the minimum number of processors needed for each task in Γ is required. However, unlike the separate analysis and Equation 4.8 which can be computed in constant time, finding the total number of processors using the joint analysis is more complicated and cannot be computed in constant time. To find the required number of processors for a single task $G \in \Gamma$, both L^q_j and W^f_j needs to be computed for all paths and q from 0 to f faults, resulting in $q \cdot p$ number of computations. Similarly to Equation 4.8, the computed values of L^q_j and W^f_j for specific values of q and f can be used to compute an estimate for the required number of processors, as shown in Equation 4.10.

$$m \geq \left\lceil \frac{W^f_j - L^q_j}{D - L^q_j} \right\rceil \tag{4.10}$$

As Equation 4.10 builds upon ideas of SDJ algorithm and Theorem 2, for each task and each value of q, the maximum value of the right-hand side of Equation 4.10 has to be found to account for the worst case. Therefore, for $q \in [0, f]$ for all

paths, the maximum estimate represents a safe estimate of the required number of processors for a single DAG task. The expression is shown in Equation 4.11.

$$m \geq \max_{j=1}^{p} \left\{ \max_{q=0}^{f} \left\{ \left\lceil \frac{W_j^f - L_j^q}{D - L_j^q} \right\rceil \right\} \right\} \tag{4.11}$$

Since the resulting estimate may not be an integer without using the ceiling function, Equation 4.11 introduces pessimism for non-integer values for the estimated number of required processors.

Algorithms 8 presents pseudocode for determining the sufficient schedulability of a taskset Γ in the presence of f faults, based on Equation 4.11. For each task $G_i \in \Gamma$, Algorithm 4 is used to find its paths P, each path length L, the maximum WCET of any node in each path C^L_{max}, its workload W, and the maximum WCET of any node in the task c_{max}. The required number of processors m_i of a single task G_i is computed by examining $q \in [0, f]$ for every path in G_i. The total number of required processors for all tasks in Γ is found and compared to the available number of processors m.

Algorithm 8: MDJ

Input: Taskset Γ, number of faults f, number of available processors m
Output: The sufficient schedulability of Γ under f faults

$M \leftarrow 0$
for $G_i = (V_i, E_i, D_i) \leftarrow \Gamma$ **do**
  $P, L, C^L_{max}, W, c_{max}, \_, \_ \leftarrow$ Algorithm 4(G_i, f)
  $m_{max} \leftarrow 0$
  for $j \leftarrow 1$ *to* $|P|$ **do**
    $len \leftarrow L[j]$
    $c^L_{max} \leftarrow C^L_{max}[j]$ `// maximum WCET in path` j
    $c^W_{max} \leftarrow \max_{v \in (V \setminus P[j])} \{c\}$ `// maximum WCET outside of path` j
    for $q \leftarrow 0$ *to* f **do**
      $W(f,q) \leftarrow W + (f - q) \cdot c^W_{max} + q \cdot c^L_{max}$
      $L(q) \leftarrow len + q \cdot c^L_{max}$
      $m_i \leftarrow \left\lceil \frac{W(f,q) - L(q)}{D_i - L(q)} \right\rceil$
      if $m_i > m_{max}$ **then**
        $m_{max} \leftarrow m_i$
  $M \leftarrow M + m_{max}$
  if $M > m$ **then**
    return; Γ is unschedulable with MDJ
Γ is schedulable with MDJ

Let n denote the number of tasks in Γ. MDJ has polynomial time complexity as its complexity can be expressed as $O(n|P|f)$. The parameters $P, L, C^L_{max}, W, c_{max}, c^W_{max}$ are assumed to be pre-computed and therefore do not affect the resulting complexity of MDJ.

4.6 Path-Based Minimization of Response Time Estimates

He et al. [13] propose a technique for estimating DAG task response times by isolating a specific number of paths, each assigned to a dedicated processor. Applying similar reasoning, the response time equation of Equation 4.1 can once more augmented to enable the path isolation, with the hope of tighter response time estimates compared to SDT and SDJ.

By analyzing the system as if processors were dedicated exclusively to the longest paths, He et al. [13] achieve a tighter response time bound than that of Graham's bound. It should be noted that this is only a way of interpreting their technique, rather than a runtime mechanism. A similar technique can be used to achieve fault tolerance when estimating the makespan of a single task, and when estimating the required number of processors for tasksets. He et al. [13] provide the following fault-free response-time bound for a single DAG task G assuming work-conserving scheduling on m cores, where $k \in [0, m-1]$:

$$R \leq \min_{j \in [0,k]} \left\{ L^* + \frac{W - \sum_{i=0}^{j} len(\lambda_i)}{m - j} \right\} \tag{4.12}$$

In this equation, the notation λ_i refers to what He et al. call a *generalized path* of G, which assumes a pre-computed *generalized path list* $(\lambda_i)_0^k$, where $k \in [0, m-1]$. A generalized path list is essentially any arbitrary set of paths or longest paths of a DAG.

Inspired by the work of He et al. [13], let $S(t)$ of Equation 4.13 denote the sum of the fault-free lengths of the t longest paths of a task G. The intention of introducing $S(t)$ is similar to the usage of the term $\sum_{i=0}^{j} len(\lambda_i)$ in Equation 4.12, where for every isolated path, we can subtract its affect to the total workload, while considering one less processors.

It is crucial to understand that $S(t)$ only accounts for individual nodes once, meaning that interwoven paths that share nodes do not get counted more than once. The key insight of the work of He et al. [13] is to remove a number of the longest paths, while assuming these paths are executed exclusively on a dedicated processor. Therefore, $S(t)$ should not count nodes multiple times in overlapping paths, as this would overestimate the workload. Three premises are assumed: 1. Path λ_t has length L_t; 2. $L_t \geq L_{t+1}$; 3. the path of nodes producing L_{max}^f corresponds to index q and is excluded from the computation of $S(t)$. It is excluded as it is accounted for separately from the rest of the paths of G in the analysis of this section. In Equation 4.13, the sum of the lengths of the t longest paths are computed, where the set V_i^{counted} represents the set of nodes already accounted for in the sum.

$$S(t) = \sum_{\substack{i=1 \\ i \neq q}}^{t} \sum_{v_k \in P_i} \left(c_k \mid v_k \notin V_i^{\text{counted}} \right) \text{ where } V_i^{\text{counted}} = \bigcup_{x=1}^{i-1} \lambda_x \tag{4.13}$$

Using similar reasoning to Equation 4.12, by assuming the nodes of at most $m-1$ longest paths execute exclusively on $m-1$ dedicated processors, a potentially

tighter response time estimate than that of SDT and SDJ can be found. We extend Equation 4.6 by considering the removal of t number of longest paths and considering t fewer processors where $k = \min(|P| - 1, m - 1)$ for $t \in [0, k]$.

$$R \leq \min_{t \in [0,k]} \left\{ L^f_{max} + \frac{W^f_{max} - L^f_{max} - S(t)}{m - t} \right\} \tag{4.14}$$

Inspired by the work of He et al. [13] Algorithm 9 presents pseudocode for determining the sufficient schedulability of a DAG task G in the presence of f faults. The algorithm is denoted SDP (Single DAG Path-based).

Algorithm 9: SDP

Input: $G = (V, E, D)$, number of faults f, number of available processors m
Output: The sufficient schedulability of G under f faults

```
P, _, _, _, _, L^f_max, W^f_max ← Algorithm 4(G, f)
R_min ← ∞
k ← min(|P| − 1, m − 1)
for t ← 0 to k do
    S(t) ← Equation 4.13
    R^f ← L^f_max + (W^f_max − L^f_max − S(t)) / (m − t)
    if R^f < R_min then
        R_min ← R_f
if R_min ≤ D then
    G is schedulable with SDP
else
    G is unschedulable with SDP
```

Let $A = max(|P|, m)$. If $S(t)$ can be precomputed, then the time complexity of SDP can be expressed as $O(A)$. If not, its complexity is instead expressed as $O(A^2|V|)$.

4.6.1 Extension for scheduling of multiple DAG tasks

The extension of SDP to handle the scheduling of multiple tasks is denoted MDP (Multiple DAG Path-based) and assumes the federated scheduling paradigm.

By reordering Equation 4.14, an estimate for the required number of processors can be found. The resulting equation is notably similar to Equation 4.8.

$$m \geq t + \left\lceil \frac{W^f_{max} - L^f_{max} - S(t)}{D - L^f_{max}} \right\rceil \tag{4.15}$$

As the estimate can vary depending on how many paths are isolated, Equation 4.15 is extended to account for different values of t, which is shown in Equation 4.16.

$$m \geq \min_{t \in [0,k]} \left\{ t + \left\lceil \frac{W^f_{max} - L^f_{max} - S(t)}{D - L^f_{max}} \right\rceil \right\} \tag{4.16}$$

Algorithm 10 presents pseudocode for determining sufficient schedulability of a taskset Γ in the presence of f faults, based on Equation 4.16. For each task G_i, Algorithm 4 is used to find the paths of the task P, as well as L^f_{max} and W^f_{max}. Then, the required number of processors is found by incrementally examining the value of t for $t \in [0, min(|P|, m-1)]$. Similarly to MDT and MDJ, the estimated required number of processors M is compared to the available m.

Algorithm 10: MDP

Input: Taskset Γ, number of faults f, number of available processors m
Output: The sufficient schedulability of Γ under f faults

for $G_i = (V_i, E_i, D_i) \leftarrow \Gamma$ **do**
  $P, \_, \_, \_, \_, L^f_{max}, W^f_{max} \leftarrow$ Algorithm 4(G_i, f)
  $m_{min} \leftarrow \infty$
  $k \leftarrow \min(|P| - 1, m - 1)$
  for $t \leftarrow 0$ *to* k **do**
    $S(t) \leftarrow$ Equation 4.13
    $m_i \leftarrow t + \left\lceil \frac{W^f_{max} - L^f_{max} - S(t)}{D_i - L^f_{max}} \right\rceil$
    if $m_i < m_{min}$ **then**
      $m_{min} \leftarrow m_i$
  $M \leftarrow M + m_{min}$
  if $M > m$ **then**
    return; Γ is unschedulable with MDP
Γ is schedulable with MDP

Let n denote the number of tasks in Γ, and $A = max(P, m)$. Similarly to SDP, the time complexity of MDP can be expressed as $O(nA^2|V|)$, which is polynomial time complexity.

5
Evaluation

This chapter offers an evaluation of the schedulability tests proposed in this thesis based on randomly generated parallel tasks. Section 5.1 describes the simulation setup and how DAG tasks are generated for examining the schedulability of a single DAG task and multiple DAG tasks. Section 5.2 presents the simulation results of the work.

5.1 Simulation Setup and DAG Task Generation

The simulation setup were based on the the task generation tool of Melani et al. [3], which in turn refers to the work of Peng et al. [31]. The generation tool was modified to generate classical DAG tasks, unlike the conditional parallel DAG task examined in [3]. Melani et al. modeled the tool to generate tasks representative of realistic workloads by implementing three real parallel programs in OpenMP. The characteristics of each program were extracted in terms of their corresponding DAG task structure, which was used as references to create the task generation tool capable of reproducing similarly-structured tasks. Therefore, the experimental setup of this work similarly represents realistic workloads.

A DAG task is generated by first creating a root node, which is initially marked as a non-sink node. The structure is then expanded recursively up to a specified recursion depth by adding child nodes, which are either parallel subgraphs, a sink node, or a successive node. The probability of generating a parallel subgraph when expanding a node is determined by the parameter p_{par}. Similarly, p_{term} controls the probability of expanding a node to a sink node, where $p_{par} + p_{term} = 1$. The nodes are expanded until a maximum recursion depth r_d is reached, where the number of branches of parallel subgraphs is uniformly selected in the range of $[2, n_{par}]$. For this work, r_d was set to 2 and n_{par} was set to 5 for all simulation results. The parameter p_{add} is a probability which controls the number of random edges between subgraphs, and is set to 0.1 for all experiments.

A DAG task G is generated in the following manner:

- the WCET of each node, $c_{i,j}$, is uniformly selected as an integer in the interval of $[1, 100]$;
- based on the generated nodes, the total workload W_i is computed;
- W^f_{max} is computed using the maximum WCET of any node in the task and is found using Algorithm 4;

- L^f_{max} is found using Algorithm 4;
- all paths are computed, including each path length and each path's corresponding largest WCET using Algorithm 4.

The simulation setup aims to test the proposed schedulability tests performance on both single DAG tasks and tasksets. The schedule parameters, T and D, are vital in this evaluation and their generation differs between the single task case versus the taskset case. The parameters are generated as follows for the single-task case:

- the period of each task, T, depends on W^f_{max} and the target utilization U, where it is generated by: $T = W^f_{max}/U$;
- the deadline is set to be implicit, $D = T$.

For the taskset case, computation of the schedule parameters, T_i and D_i instead follows the approach of He et al. [13], where the schedule parameters are generated as follows:

- the parameter α is randomly selected in $[0, 0.25]$;
- the period, T, is computed by $T = L^f_{max} + \alpha(W^f_{max} - L^f_{max})$, which therefore lies in the range of $[L^f_{max}, \frac{3L^f_{max}+W^f_{max}}{4}]$;
- the deadline is randomly selected within the range of $[L^f_{max}, T]$. Therefore, a task deadline can lie in the range of $[L^f_{max}, \frac{3L^f_{max}+W^f_{max}}{4}]$.

An important difference in this work, compared to the period and deadline generation approach of He et al., is the use of the fault-dependent values L^f_{max} and W^f_{max} instead of the original L and W from [13]. Since fault recovery increases the total workload in the system, L^f_{max} and W^f_{max} are necessary to avoid underestimating task utilization.

The α parameter directly impacts the resulting utilization of tasks. For values of $\alpha < 1$, the minimum generated utilization of tasks is 1, meaning only heavy tasks are generated. As the examination of how faults affect heavy tasks is of interest in this work, α was chosen such that at most one light task per taskset was generated. This was done by continuously generating heavy tasks until the total utilization of the taskset exceeded the target utilization. The final task's period was adjusted such that it matched the remaining utilization, which could either be another heavy task, or a light task. Only a single task was generated for $U \leq 1$, to which the period generated using the α parameter was scaled to fit the current target utilization, essentially meaning that α does not impact these scenarios. Since a maximum of one light task could be generated, the light tasks required at most one processor.

One exception was made in the case of examining the effects of varying the number of tasks per taskset. In this scenario, task deadlines were set to be implicit ($D_i = T_i$). To find the number of processors required by multiple light tasks, it first requires an ordering of the tasks when assigning them to a processor. The tasks were assigned according to decreasing utilization, meaning the task with the highest utilization was assigned first. As the required number of processors dynamically

increases as more light tasks are considered, their order was inherent to when they were added. The uniprocessor schedulability test of EDF priority was assumed, as its utilization bound test is easy to implement and when paired with implicit deadlines has a utilization bound of 1 [8], which is higher than other alternatives. In other words, when examining if another light task fits in a processor, the task fits if the total task utilization of all tasks executing on that processor does not exceed 1. If a task does not fit on a specific processor, then the next existing processor is examined similarly. If no existing processor can fit the light task, another processor is assumed to be required, to which it is assumed the task then fits, as its utilization is less than 1. Here, light tasks can together require more than one processor.

For the single task case, the SDT, SDJ, and SDP algorithms were applied to each generated task, determining schedulability based on whether their produced response time estimate exceeded the task's deadline or not. For the taskset case, the schedulability of tasksets was determined based on the estimate of the required number of processors produced by the MDT, MDJ, and MDP algorithms. Here, the estimated required number of processors for the heavy and light tasks of each taskset was computed and compared to the available number of processors. An estimate exceeding the available number of processors implies that schedulability cannot be guaranteed by the algorithm producing the estimate.

5.2 Simulation Results

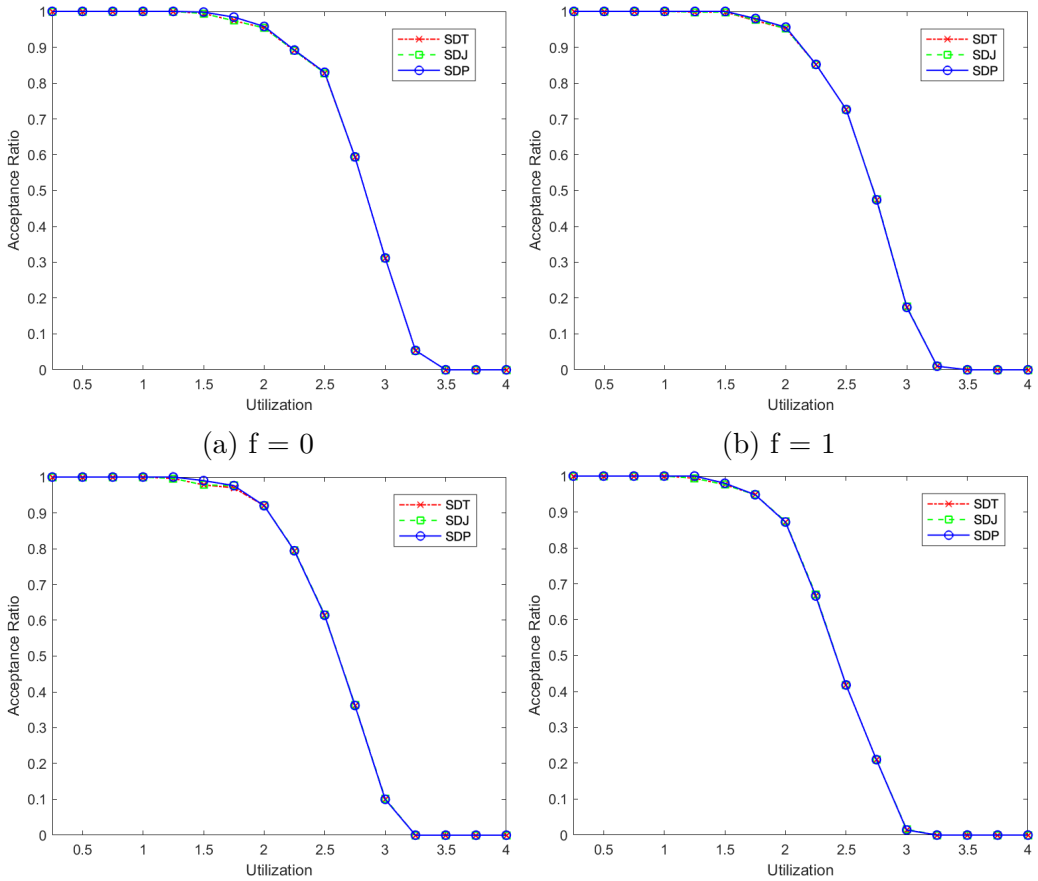
Based on the simulation setup, the main metric used to judge performance is the *acceptance ratio*. The acceptance ratio is the fraction of the total number of randomly generated tasksets guaranteed to be schedulable according to a particular schedulability test. In the rest of this section, we will be presenting the acceptance ratio of our proposed algorithms, varying different parameters such as the total system utilization U, the number of available processors m, and the number of tasks in each taskset n. To generate task utilization in the case of varying the number of tasks, the UUnifast [32] algorithm was used. UUnifast is used to split the fixed system utilization U into the n utilization values corresponding to the utilization of each of the n tasks. In all other cases, the number of tasks in each taskset was unknown, and UUnifast was therefore not used.

When examining single DAG tasks, a total of 500 tasks were generated per utilization level U, where $U \in [0.25, 0.5, ..., m]$. Notably, this refers to 500 individual tasks, not 500 tasksets, as this case aimed to test the acceptance ratio when the individual tasks become increasingly heavier. For scenarios examining the scheduling of multiple tasks, 2000 tasksets were generated per utilization level for $U \in [0.25, 0.5, ..., m]$, where new tasks were progressively added to each set until the total utilization was equal to the target utilization U. The period of the last task was adjusted such that the total utilization equals U.

5.2.1 Single DAG task performance

The first scenario examines the single-task makespan estimation algorithms' performance on a single DAG task, where a total of 500 tasks are generated at each

utilization level. The upper limit for the total system utilization is set to the available number of processors for each scenario. Figure 5.1 presents the acceptance ratios of SDT, SDJ, and SDP when the number of available processors $m = 4$ and the total system utilization $U \in [0.25, 0.5, ..., 4]$ for $f = 0, 1, 2, 4$. The four subfigures share the trend of maintaining 100% acceptance ratio for utilization values lower than 1, then tapering towards 0% around $U = 3$ to $U = 3.5$.



(a) f = 0 (b) f = 1

(c) f = 2 (d) f = 4

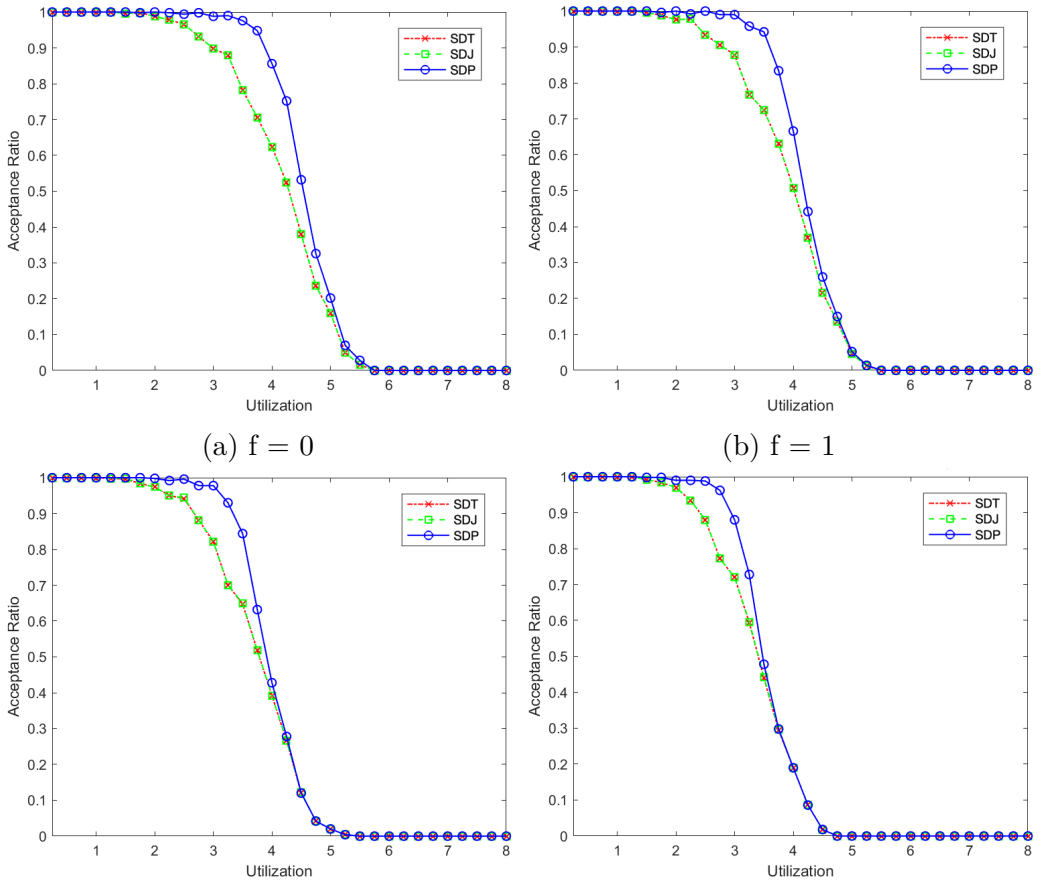
Figure 5.1: Acceptance ratio of SDT, SDJ, and SDP where $m = 4$ and $f = 0, 1, 2, 4$ for 500 tasks and $U \in [0.25, 0.5, ..., 4]$ (Implicit deadlines).

The three algorithms perform nearly identically for $U = m = 4$, showing a predominant overlap, except for the slight advantage of SDP at around $U = 1.5$ to $U = 1.75$, varying depending on the number of faults. Across the four subfigures, the SDT and SDJ algorithms seem to completely overlap. In other words, either both tests produce a response time estimate smaller than the deadline and therefore both meet the deadline, or both fail to meet the deadline. The similar performance across all three algorithms can be attributed to the relatively low values of U and m. Similarly to the approach of He et al. [13], the SDP algorithm takes advantage of multiple long paths of a task, which SDT and SDJ do not. The potential improvements of SDP over SDT and SDJ cannot be seen in this case, and the similarities are likely due to the simulation parameters.

The sigmoid-like shape of the three curves in each subfigure can partly be attributed to the simulation setup and approach to evaluating the algorithms. In general, the acceptance ratio decreases as total system utilization increases. As the load of the system increases it becomes more difficult to schedule. Additionally, as the U increases, the period and implicit deadline decrease due to how the period is generated. Therefore, for larger values of U the acceptance ratio decreases as the estimated response time is more likely to be larger than the relatively shorter deadline.

Figure 5.2 instead shows the results of SDT, SDJ, and SDP for $m = 8$, $U \in [0.25, 0.5, ..., m]$, and $f = 0, 1, 2, 4$ for 500 tasks per utilization level. In contrast to $m = 4$, for $m = 8$ the SDP algorithm now outperforms both SDT and SDJ for all values of f in a limited interval of U, where the interval varies with f. For $f = 0$, SDP outperforms the others between $U = 1.75$ and $U = 5.75$, where the largest difference in acceptance ratio can be seen for $U = 3.75$ for $f = 0$, where SDP has roughly 25% greater acceptance ratio than the overlapping SDT and SDJ. As the number of faults increases, the difference in acceptance ratio between the algorithms in the observed interval becomes smaller. Similarly, the algorithms tend to perform worse as the number of faults increases, where they reach 0% acceptance ratio at around $U = 5.75$ for $f = 0$ and $U = 4.75$ for $f = 4$. The intermediary steps of $f = 1$ and $f = 2$ show a similar progression, where each graph is slightly shifted leftward moving from $f = 1$ to $f = 2$.

SDP is most resilient to faults of the three algorithms, as can be seen at $U = 2.75$. Comparing their performance when $f = 0$ to $f = 4$, SDP drops only around 4% in acceptance ratio, whereas SDT and SDJ instead drop around 16%. As the number of faults increases, the inherent pessimism of each algorithm becomes more pronounced, further impacting the acceptance ratio. SDP offers better or the same performance as the others, regardless of the number of faults. This can be attributed to its inclusion of accounting for multiple long paths of each task. If SDP did not account for long paths other than the one producing L^f_{max}, i.e. if $j = 0$, it would produce the same estimate as SDT. Therefore, Figure 5.2 highlights how SDP is less pessimistic than SDT and SDJ.



(a) f = 0 (b) f = 1

(c) f = 2 (d) f = 4

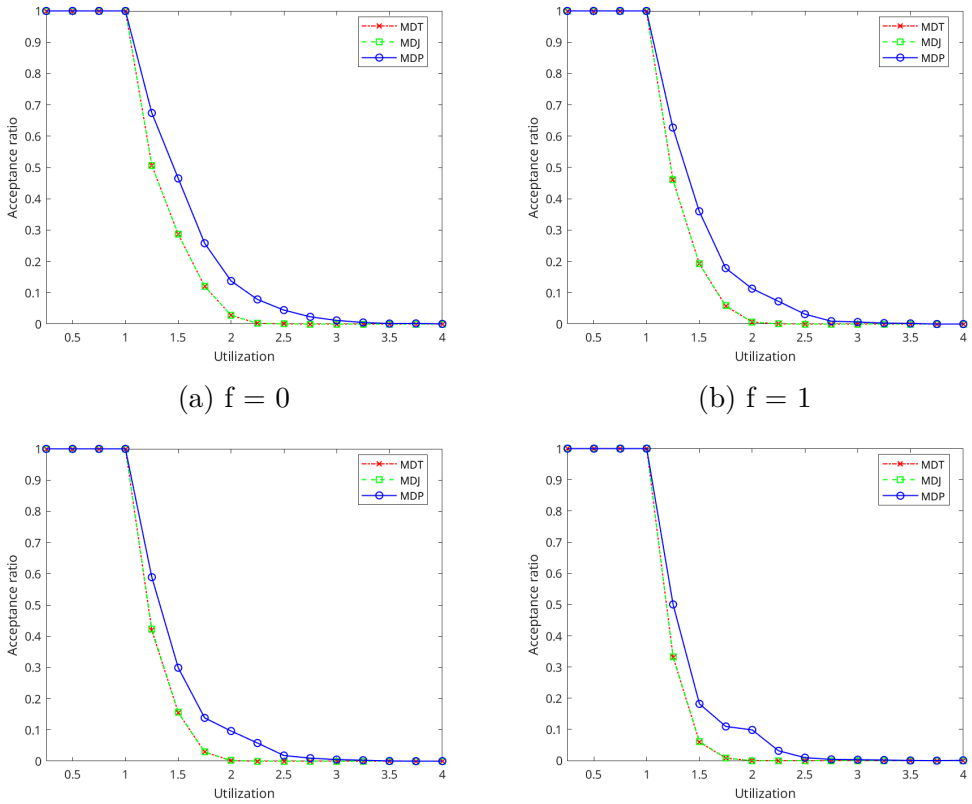
Figure 5.2: Acceptance ratio of SDT, SDJ, and SDP where $m = 8$ and $f = 0, 1, 2, 4$ for 500 tasks and $U \in [0.25, 0.5, ..., 8]$ (Implicit deadlines).

5.2.2 DAG taskset performance

Each of the following sections presents the performance of the MDT, MDJ, and MDP algorithms on 2000 tasksets. A vital difference from the single DAG performance evaluation of Section 5.2.1 is how task deadlines and task periods were being generated, which instead used a similar method to that of He et al. [13]. However, the results presented in Figure 5.10 and Figure 5.11, instead use implicit deadlines and the same schedule parameter generation of Section 5.2.1.

The first scenario examines how the total system utilization U impacts performance. Figure 5.3 shows a scenario where $m = 4$ and $U \in [0.25, 0.5, ..., 4]$ for $f = 1, 2, 3, 4$. All four subfigures show that for $U \leq 1$, all algorithms achieve 100% acceptance ratio. As only a single light task can be generated for $U \leq 1$, and that light tasks are handled the same for the three algorithms, the four available processors can always guarantee schedulability of the task. However, the number of tasks generated at each utilization level is not fixed for $U > 1$, and can therefore vary. Heavy tasks can only be generated in tasksets with total utilization $U > 1$. Furthermore, as the available number of processors, m, is relatively high compared to the utilization when $U \leq 1$, it is reasonable that each algorithm can guarantee

schedulability for all 2000 tasksets, regardless of the number of faults.



(a) f = 0 (b) f = 1

(c) f = 2 (d) f = 4

Figure 5.3: Acceptance ratio of MDT, MDJ, and MDP where $m = 4$ and $f = 0, 1, 2, 4$ for 2000 tasksets per utilization level $U \in [0.25, 0.5, ..., 4]$ (Constrained deadlines).

In general, the acceptance ratio decreases as the f increases, and the slope of each algorithm in Figure 5.3 becomes increasingly steep, pushing data points that lie close to 0% acceptance ratio for $f = 0$ even nearer 0% for $f = 4$. When $U > 1$ the acceptance ratio for the three algorithms all start to decline across all subfigures as U increases, and MDP in general outperforms or performs the same as MDT and MDJ for all values of f.

MDT and MDJ perform similarly, showing a clear overlap across the four subfigures of Figure 5.3. However, MDJ can potentially outperform MDT depending on the characteristics of the generated DAG tasks. MDJ extends MDT by accounting for additional corner cases, providing a less pessimistic estimate for these scenarios. In general, the DAG structures to which MDJ outperforms MDT are rarely generated by the simulation setup, which is therefore reflected in terms of their overlap in each figure.

Regardless of the number of faults, the performance gain of MDP over MDT and MDJ can likely be attributed to similar factors producing the performance increase of SDP over SDT and SDJ in the single task case. MDP sees a significant performance increase by accounting for multiple long paths in each task, which is the key point of the work of He et al.[13], which reduces the required number of

processors for each taskset and therefore increases the acceptance ratio.

Notably, a visible step in Figure 5.3 of MDP forms around $U = 2$ as the number of faults increases, where the acceptance ratio drops sharply for $U \geq 2.25$. A plausible explanation relates to the range of possible task utilization values in relation to the total system utilization. Figure 5.4 shows two key relationships of Figure 5.3: the average number of tasks per taskset and the average task utilization per taskset across the different utilization levels. For $1 < U \leq 2$, the graphs reveal a transition: while the average task utilization lies around 1, the average number of tasks increases to 2. This indicates a shift from generating tasksets predominantly containing a single heavy task to generating tasksets each with one heavy and one light task. As the system utilization approaches 2, tasksets contain at most one heavy task, accompanied by a light task of increasing utilization. Consequently, the similar acceptance ratio of MDP for $U = 1.75$ to $U = 2$ in Figure 5.3 likely stems from this generation pattern. Similarly, the sharp drop at the subsequent $U = 2.25$ can be attributed to the total system utilization now being large enough to generate two heavy tasks.

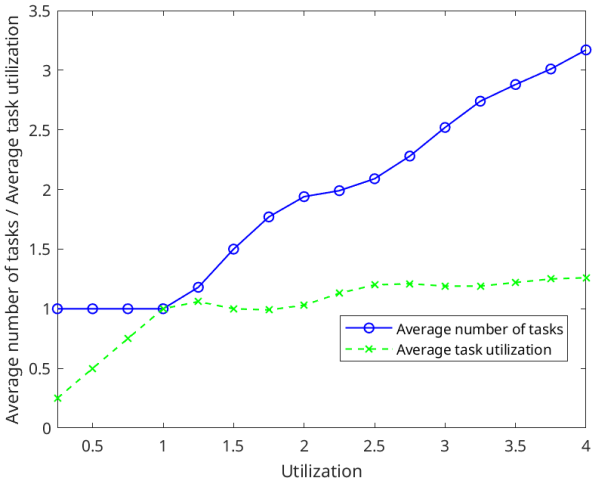
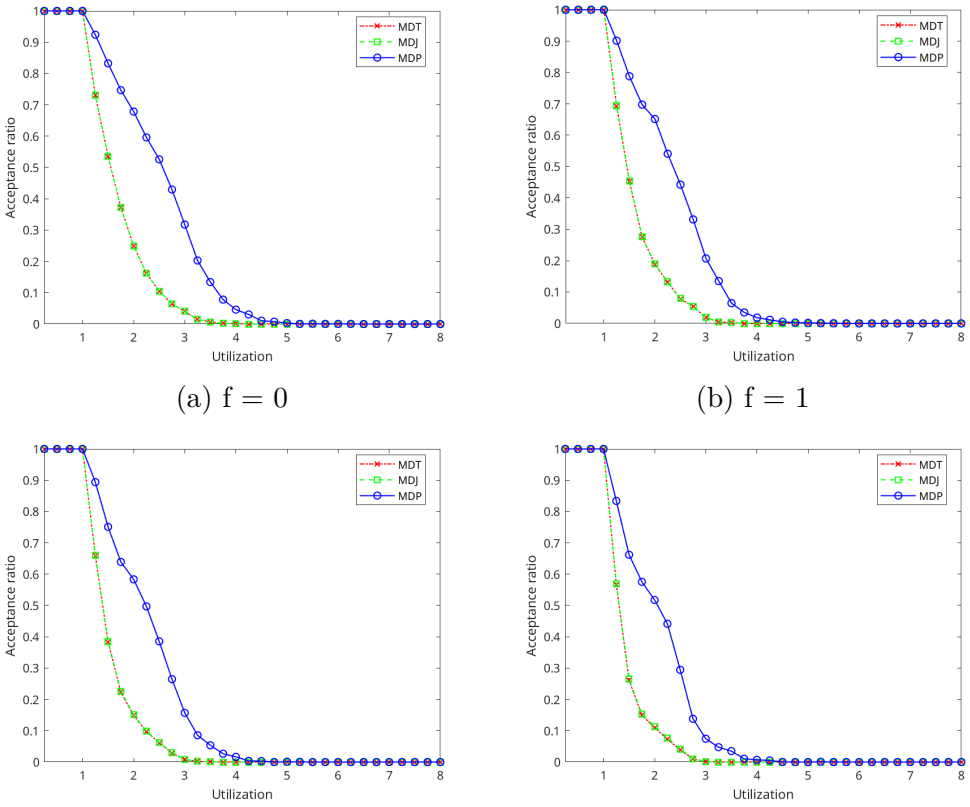


Figure 5.4: Average number of tasks and average task utilization per utilization level for Figure 5.3.

Figure 5.5 shows a scenario where $m = 8$ and $U \in [0.25, 0.5, ..., 8]$ for $f = 0, 1, 2, 4$. It shows the same general trend as seen in Figure 5.3 of MDP performing the same or outperforming MDT and MDJ. However, the performance gains of MDP are now further accentuated. MDP benefits more than MDT and MDJ from the increase in the number of processors compared to the results of Figure 5.3. This benefit gradually diminishes as the number of processors increases further, at different rates depending on the value of U, as later shown in Figure 5.8 and Figure 5.9. A similar bump as in Figure 5.3 can be seen around $U = 2$ as f increases and likely occurs for the same reason.

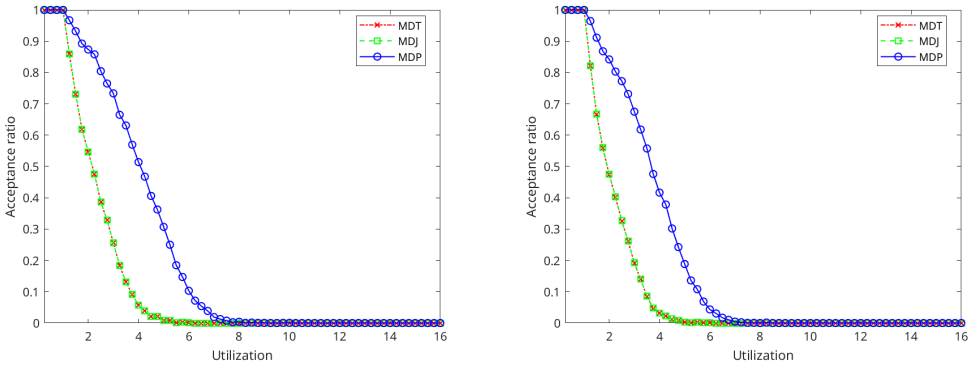


(a) f = 0 (b) f = 1

(c) f = 2 (d) f = 4

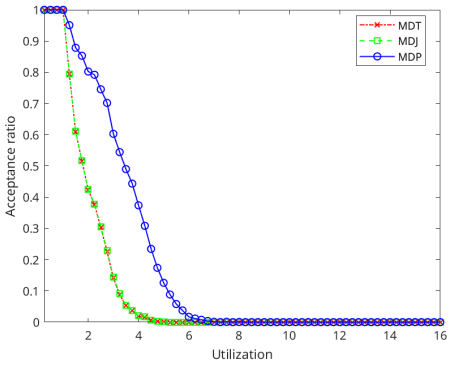
Figure 5.5: Acceptance ratio of MDT, MDJ, and MDP where $m = 8$ and $f = 0, 1, 2, 4$ for 2000 tasksets per utilization level $U \in [0.25, 0.5, ..., 8]$ (Constrained deadlines).

Figure 5.6 presents the result of MDT, MDJ, and MDP, when $m = 16$ and $U \in [0.25, 0.5, ..., 16]$ for $f = 0, 1, 2, 4$. The performance increase of MDP becomes slightly more evident, with greater improvements in the acceptance ratio for specific values of U. For $m = 16$, MDP clearly outperforms MDT and MDJ for $U \in [1, 6]$ across the four subfigures, benefiting more from the increased number of available processors. This indicates that for this range of U, MDP better utilizes the multicore platform, especially in the presence of faults.

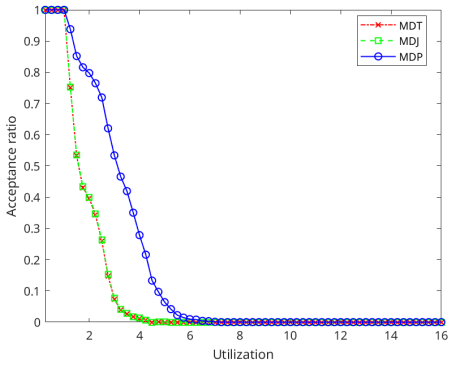


(a) f = 0

(b) f = 1



(c) f = 2



(d) f = 4

Figure 5.6: Acceptance ratio of MDT, MDJ, and MDP where $m = 16$ and $f = 0, 1, 2, 4$ for 2000 tasksets per utilization level $U \in [0.25, 0.5, ..., 16]$ (Constrained deadlines).

To highlight the sensitivity to faults of the best-performing MDP test for $m = 16$ and when a total system utilization of $U = 16$, Figure 5.7 presents the performance of MDP for the different values of f in Figure 5.6.

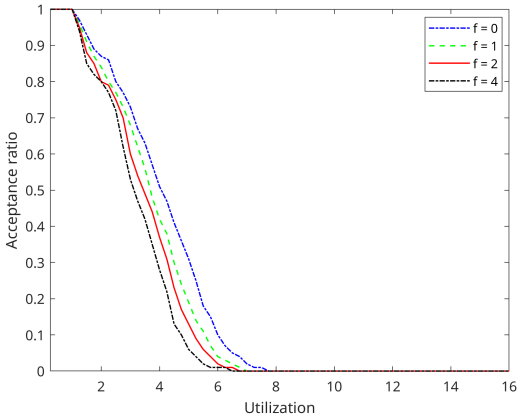


Figure 5.7: Acceptance ratio of MDP when $m = 16$ and $f = 0, 1, 2, 4$ for 2000 tasksets per utilization level $U \in [0.25, 0.5, ..., 16]$ (Constrained deadlines).

Figure 5.7 explicitly indicates the impact of an increased number of faults. When

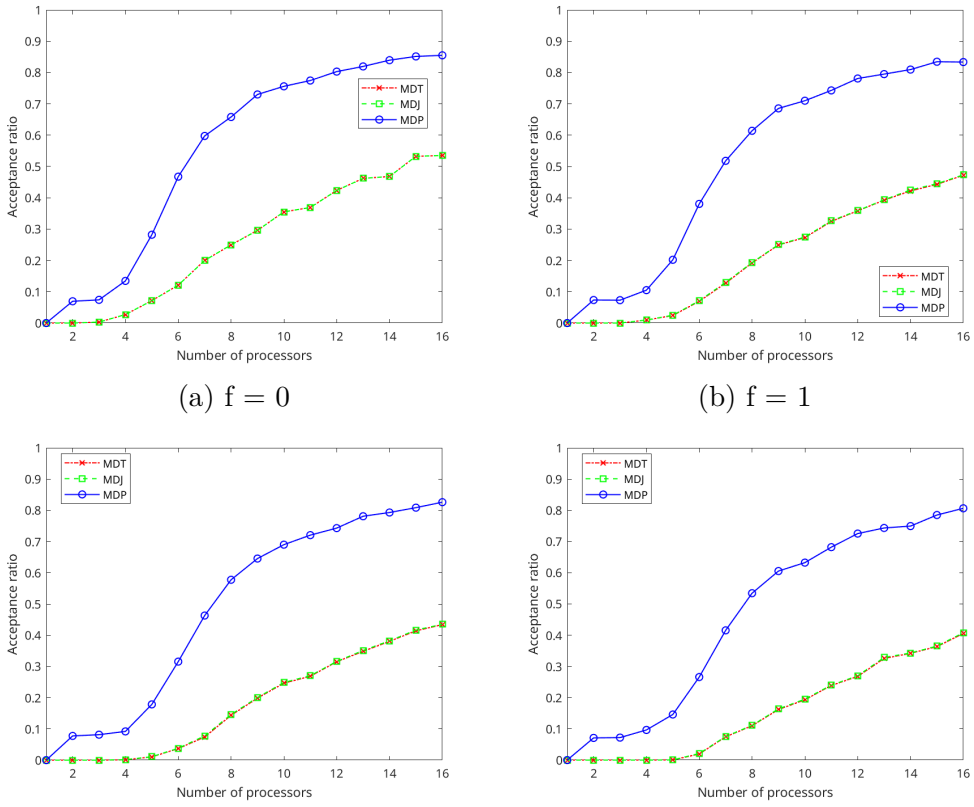
f increases, each graph is moved leftward with a steeper slope at times. This indicates a worse acceptance ratio for the same value of U, which follows the intuition of fault tolerance coming at the cost of sacrificing some computational power.

5.2.3 Varying the number of processors

By varying the number of processors, m, and keeping the total system utilization fixed, the impact of m on schedulability for MDT, MDJ, and MDP can be visualized and examined. Figure 5.8 shows the scenario where $U = 2$ and $m \in [1, 2, ..., 16]$ for $f = 0, 1, 2, 4$. In general, as the number of processors increases, so does the acceptance ratio for all three algorithms regardless of the number of faults. The relatively low value of U in this scenario likely contributes to the faults' low impact on performance, as the acceptance ratio only gradually declines as f increases.

The performance improvement of MDP over MDT and MDJ remains evident regardless of the number of faults. The performance gap grows with an increased number of available processors, to which it gradually tapers for larger values of m, varying depending on the value of f. This characteristic can likely be attributed to the diminishing returns of an increased number of processors when scheduling tasksets with a relatively low total utilization of $U = 2$. On the other hand, when m grows larger, the larger ratio between m and U should imply better performance. Yet, regardless of the number of faults, no algorithm reached 100% schedulability. This behavior appears to be common in similar works. Even though there are notable differences, and the results cannot be directly compared, Melani et al. [3] show a similar result when varying the number of processors and keeping the utilization fixed in a fault-free scenario. More specifically, the algorithms' inability to reach 100% acceptance ratio appears to be related to the usage of constrained deadlines.

The simulation variable α is randomly generated in $[0, 0.25]$, and for α values close to 0, the resulting deadline approaches L^f_{max}. This results in the task becoming harder to schedule, and likewise, the estimate of the required number of processors for said task increases. Therefore, it is likely that some of the 2000 tasksets are unschedulable according to our algorithms due to this reason, even if a relatively higher number of processors is available.



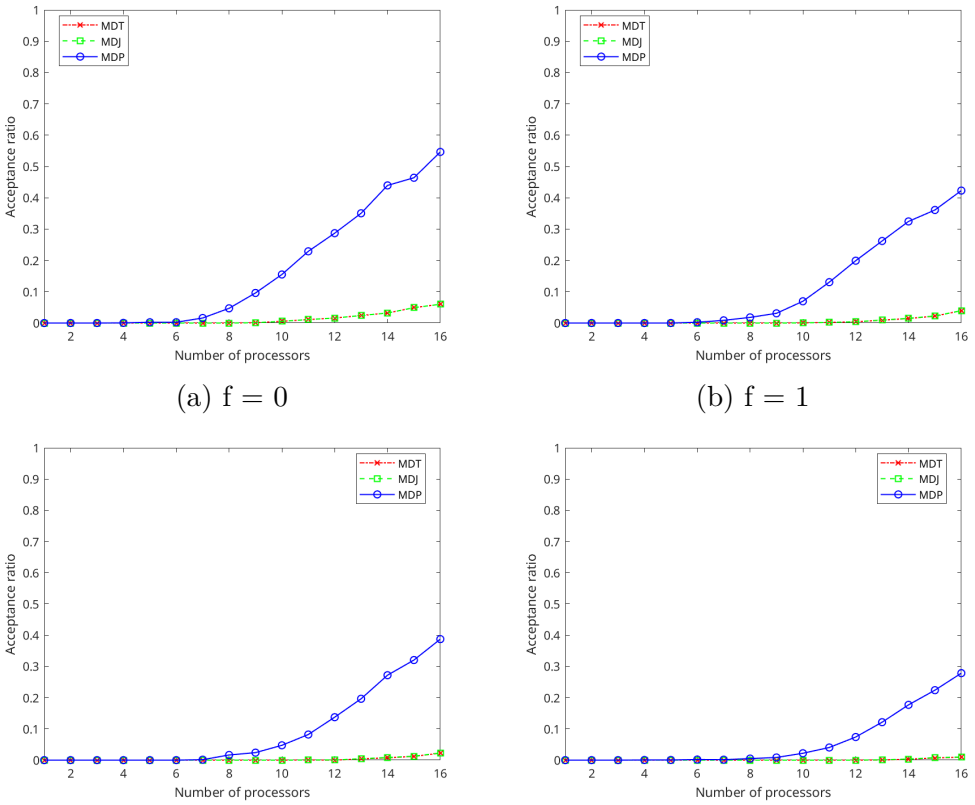
(a) f = 0 (b) f = 1

(c) f = 2 (d) f = 4

Figure 5.8: Acceptance ratio of MDT, MDJ, and MDP when varying the number of processors m, where $U = 2$, $f = 0, 1, 2, 4$ and 2000 tasksets for each value of $m \in [1, 2, ..., 16]$ (Constrained deadlines).

Figure 5.9 instead shows a scenario where the fixed system utilization is $U = 4$. Now, faults greatly impact the performance of the three algorithms in comparison to Figure 5.8. Furthermore, as m grows, an explicit difference in the rate of change between MDP and the overlapping MDT and MDJ can be seen. This is likely explained by U becoming large enough to better showcase the benefit of exploiting multiple long paths of a DAG task, which is employed in the case of MDP.

Section 5.2.2 showed that as the number of processors and the varying system utilization increased, the gains of MDP over MDT and MDJ grew larger for specific intervals. Similarly, Figure 5.9 indicates that the performance of MDP benefits more from an increase in the number of available processors, as more paths can be kept separate.



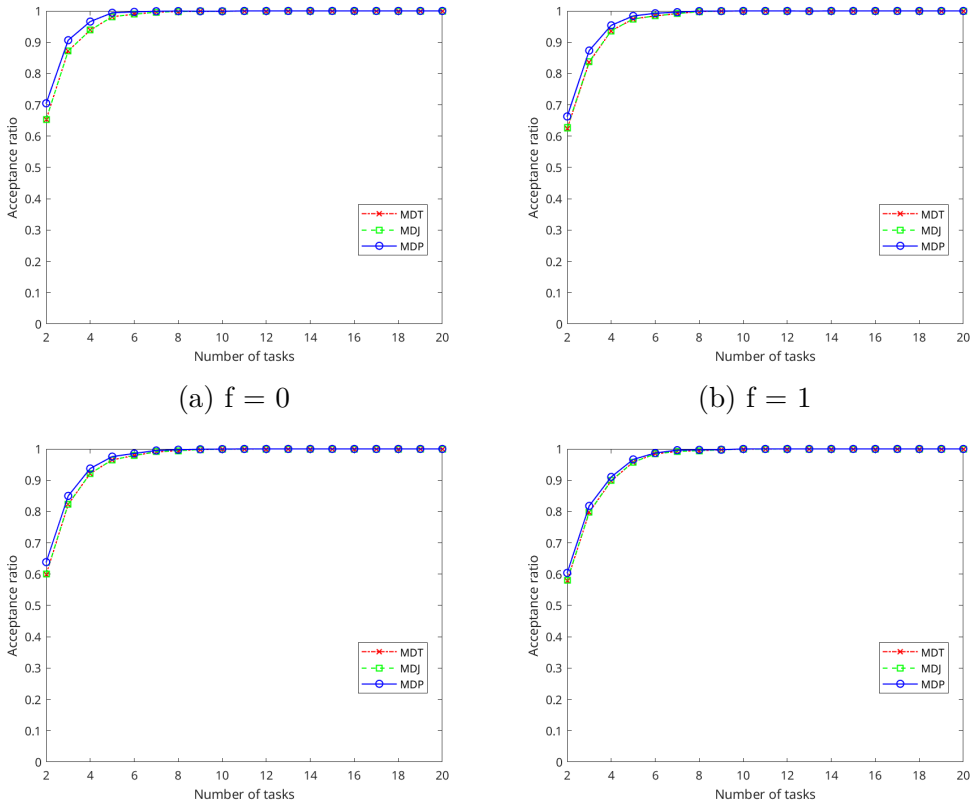
(a) f = 0 (b) f = 1

(c) f = 2 (d) f = 4

Figure 5.9: Acceptance ratio of MDT, MDJ, and MDP when varying the number of processors m, where $U = 4$, $f = 0, 1, 2, 4$ and 2000 tasksets for each value of $m \in [1, 2, ..., 16]$ (Constrained deadlines).

5.2.4 Varying number of tasks in taskset

For a given system utilization U, the number of tasks in a taskset affects how difficult it is to schedule. Melani et al. [3] speak about the intuition of it being easier to schedule a large number of light tasks as opposed to a smaller number of heavy tasks. Figure 5.10 appears to reflect a similar sentiment, where it shows a trend of increasingly higher acceptance ratios as the number of tasks increases, regardless of the number of faults. The similar performance of MDT, MDJ, and MDP, can likely be attributed to relatively low values of U and m. The usage of implicit deadlines further accentuates this point, as the implicit deadlines generated in by the simulation setup are larger in general, compared to the constrained case of prior sections.



(a) f = 0 (b) f = 1

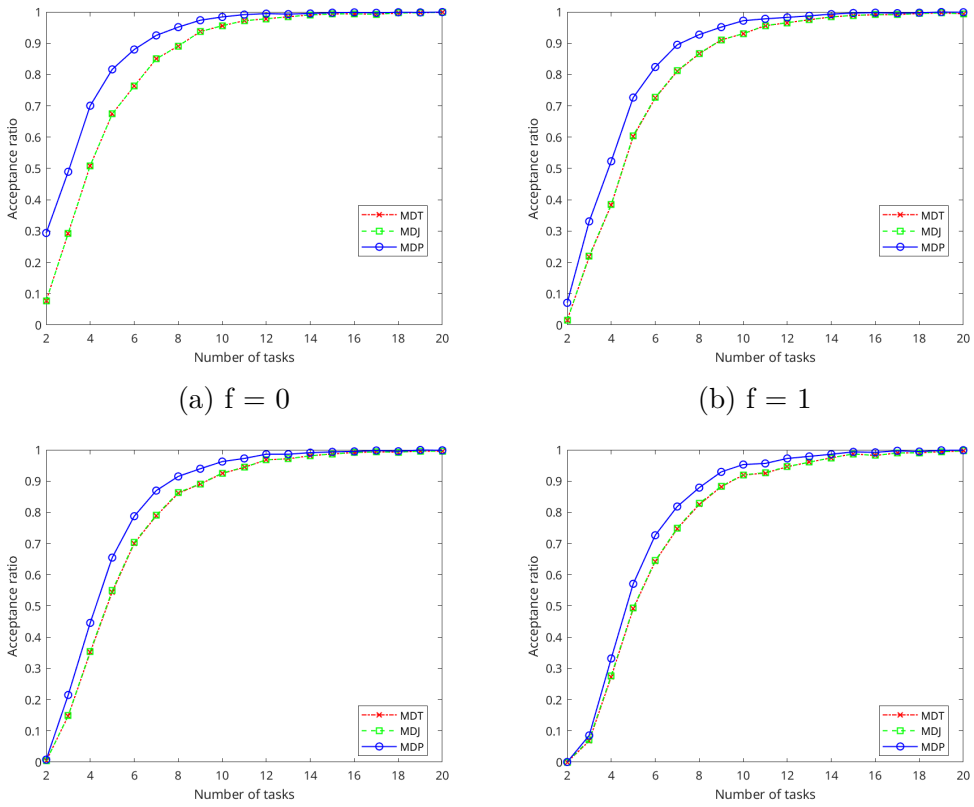
(c) f = 2 (d) f = 4

Figure 5.10: Acceptance ratio of MDT, MDJ, and MDP when varying the number of tasks n, where $U = 2$, $m = 4$, $f = 0, 1, 2, 4$ and 2000 tasksets for each value of $n \in [2, 3, ..., 20]$ (Implicit deadlines).

Figure 5.11 instead shows a scenario where both U and m have been doubled to 4 and 8 respectively. The algorithms' performances now start to differentiate, and as the number of faults increases, the acceptance ratio decreases. As n grows, each of the 2000 tasksets must split the fixed system utilization between a greater number of tasks, enabling more light tasks to be generated. If techniques for allowing light tasks to execute in the slack of heavy tasks are not employed (which is the case of this work), heavy tasks might not fully utilize each of their dedicated processors. In

other words, the estimated required number of processors for a heavy task might introduce slack. When n grows large enough, it seems as if this slack is instead utilized by the many generated light tasks.

As the increased U between Figure 5.10 and Figure 5.11 leads to worsened performance for each algorithm—even when m is increased by the same factor—it speaks to the intuition highlighted by Melani et al., as the reduced performance is seen for low values of n where tasksets containing a relatively larger number of heavy tasks are more likely.



(a) f = 0 (b) f = 1

(c) f = 2 (d) f = 4

Figure 5.11: Acceptance ratio of MDT, MDJ, and MDP when varying the number of tasks n, where $U = 4$, $m = 8$, $f = 0, 1, 2, 4$ and 2000 tasksets for each value of $n \in [2, 3, ..., 20]$ (Implicit deadlines).

Permanent faults are not considered in the evaluation, nor have they been considered in the algorithms presented in Chapter 4. Permanent faults are of great importance, but transient faults became the main focus of this thesis. Similarly to transient faults, recovery from permanent faults can be introduced by through re-execution and consider one less processor per permanent fault.

This chapter has presented the results of the proposed schedulability tests under varying system parameters, with a particular focus on the number of faults. The findings demonstrate that the SDP and MDP algorithms, which leverage information from multiple long paths of a DAG, consistently perform at least as well as, and often outperform, the other approaches across different scenarios. The SDT and SDJ

algorithms performed nearly identically, much like MDT and MDJ, which was an unexpected outcome. This is likely due to the set of possible DAG structures generated, which did not emphasize the differences between them. Notably, the MDP algorithm benefits significantly from an increased number of available processors, seeing larger performance gains than MDT and MDJ. Varying the number of tasks in the generated tasksets had little impact on the resulting acceptance ratio.

6
Conclusion and Future Work

This work aimed to provide methods of examining a real-time system modeled as DAG tasks when considering fault tolerance. We have proposed a scheduling algorithm and its analysis to present schedulability tests, where the fault-tolerant schedulability analysis of single DAG tasks and tasksets was presented in terms of six tests. The SDT, SDJ, and SDP tests relate to the schedulability of a single DAG task using response time estimates. The MDT, MDJ, and MDP tests relate to determining the schedulability of tasksets under federated scheduling, using estimates for the required number of processors. We have proposed methods for handling a finite number of transient faults within a given interval, proposing a model that accounts for fault tolerance through re-execution of nodes. Utilizing existing scheduling analysis of parallel DAG tasks as a foundation, the goal of minimizing pessimism is to provide schedulability tests that, in cases where a taskset is schedulable on the system, will reliably indicate its schedulability. This is to ensure a better prediction of whether deadlines are met or missed, rather than producing conservative estimates and succumbing to analysis pessimism that might falsely reject feasible tasksets.

The combination of the parallel DAG task model and fault tolerance highlighted the algorithms' sensitivity to changes in task periods and deadlines. A significant portion of time of this work was dedicated to finding ways of generating these schedule parameters while considering fault tolerance. The aim was to find ways of exemplifying the algorithms' relative performance and unique strengths. The existing methods of generating task deadlines and task periods utilized by He et al.[13] and Melani et al. [3] is done in a fault-free scenario. In this work, these methods were examined for both single task and taskset scheduling. However, as faults affect a task's fault-free utilization, modifications to the parameter generation were necessary to provide accurate results. Thus, a potential improvement lies in finding better ways of generating these parameters.

Common outcomes in the simulation results were often of two scenarios when comparing the acceptance ratio of the tests: 1. The tests produce response time estimates that either both meet the task's deadline (or produce an estimate less than the number of available processors); 2: Or, they produce estimates that fail to meet the requirements in terms of deadline or number of available processors. However, this was not always accompanied by the tests producing the same response time estimate or estimate of the required number of processors. For example, the MDP test often produced lower estimates than MDT and MDJ, but could not always be reflected in terms of acceptance ratio if it occurred under scenario 2. In other words, the choice of deadline directly impacts the resulting acceptance ratio of the tests,

and it could be argued that a different method of generating the schedule parameters could result in the tests performing differently. As a result, the choice of deadline generation for fault-tolerant DAG task scheduling could be explored further, or at the very least be considered when designing simulation setups for evaluation.

The MDP algorithm was derived from the work of He et al. [13] to address a fault-tolerant scenario. It could be argued that the performance of MDP could be improved by implementing an algorithm more closely following theirs. For this work, time restraints limited such an exploration.

Understanding how faults impact the execution sequence and schedulability of a DAG task is complex. Exploring how fault tolerance impacts the exploitation of the DAG task structure is a potential future research area.

In this work, the problem of examining an exponentially growing number of fault scenarios for parallel DAG tasks is reduced to polynomial time complexity algorithms. By combining fault tolerance with the parallel DAG model, this work showcased that better leveraging a DAG task's structural information leads to improved acceptance ratio performance. The SDP and MDP algorithms, which leveraged such information, generally outperformed the others for the same scenario and provided polynomial time complexity. As a result, such exploitation enables system designers to identify when a system requires fewer resources, leading to more efficient provisioning of system resources.

Bibliography

[1] OpenMP Architecture Review Board, *OpenMP application program interface version 5.2*, 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[2] R. M. Pathan, P. Voudouris, and P. Stenström, "Scheduling parallel real-time recurrent tasks on multicore platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 915–928, 2018. DOI: 10.1109/TPDS.2017.2777449.

[3] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Schedulability analysis of conditional parallel task graphs in multicore systems," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 339–353, 2017. DOI: 10.1109/TC.2016.2584064.

[4] B. Gutjahr and M. Werling, "Automatic collision avoidance during parking and maneuvering an optimal control approach," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, 2014, pp. 636–641. DOI: 10.1109/IVS.2014.6856575.

[5] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *2012 IEEE 33rd Real-Time Systems Symposium*, 2012, pp. 63–72. DOI: 10.1109/RTSS.2012.59.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95, Santa Barbara, California, USA: Association for Computing Machinery, 1995, pp. 207–216, ISBN: 0897917006. DOI: 10.1145/209936.209958. [Online]. Available: https://doi.org/10.1145/209936.209958.

[7] R. M. Pathan, "Fault-tolerant and real-time scheduling for mixed-criticality systems," *Real-Time Systems*, May 2014. [Online]. Available: https://doi.org/10.1007/s11241-014-9202-z.

[8] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973, ISSN: 0004-5411. DOI: 10.1145/321738.321743. [Online]. Available: https://doi.org/10.1145/321738.321743.

[9] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, Oct. 2011, ISSN: 0360-0300. DOI: 10.1145/1978802.1978814. [Online]. Available: https://doi.org/10.1145/1978802.1978814.

[10] R. M. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *2012 24th Euromicro Conference on Real-Time Systems*, 2012, pp. 309–320. DOI: 10.1109/ECRTS.2012.29.

[11] X. Jiang, H. Liang, N. Guan, Y. Tang, L. Qiao, and Y. Wang, "Scheduling parallel real-time tasks on virtual processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 33–47, 2023. DOI: 10.1109/TPDS.2022.3213024.

[12] X. Jiang, N. Guan, X. Long, and H. Wan, "Decomposition-based real-time scheduling of parallel tasks on multicores platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2319–2332, 2020. DOI: 10.1109/TCAD.2019.2937820.

[13] Q. He, N. Guan, M. Lv, X. Jiang, and W. Chang, "Bounding the response time of dag tasks using long paths," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 474–486. DOI: 10.1109/RTSS55097.2022.00047.

[14] R. M. Pathan and J. Jonsson, "Exact fault-tolerant feasibility analysis of fixed-priority real-time tasks," in *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2010, pp. 265–274. DOI: 10.1109/RTCSA.2010.24.

[15] F. C. Carlis Collins, "Fpgas on mars," *Xcell journal Issue 50*, pp. 8–11, 2004, ISSN: 50. [Online]. Available: http://bitsavers.informatik.uni-stuttgart.de/components/xilinx/xcell/Xcell50.pdf.

[16] R. M. Pathan and J. Jonsson, "Ftgs: Fault-tolerant fixed-priority scheduling on multiprocessors," in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, 2011, pp. 1164–1175. DOI: 10.1109/TrustCom.2011.158.

[17] G. Chen, N. Guan, K. Huang, and W. Yi, "Fault-tolerant real-time tasks scheduling with dynamic fault handling," *Journal of Systems Architecture*, vol. 102, p. 101 688, 2020, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.101688. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762119304953.

[18] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969. DOI: 10.1137/0117039. eprint: https://doi.org/10.1137/0117039. [Online]. Available: https://doi.org/10.1137/0117039.

[19] S. Baruah and A. Marchetti-Spaccamela, "Feasibility Analysis of Conditional DAG Tasks," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, B. B. Brandenburg, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 196, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 12:1–12:17, ISBN: 978-3-95977-192-4. DOI: 10.4230/LIPIcs.ECRTS.2021.12. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2021.12.

[20] 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.

[21] N. Ueter, G. von der Brüggen, J.-J. Chen, J. Li, and K. Agrawal, "Reservation-based federated scheduling for parallel real-time tasks," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 482–494. DOI: 10.1109/RTSS.2018.00061.

[22] R. Pathan, "Real-time scheduling algorithm for safety-critical systems on faulty multicore environments," *Real-Time Systems*, vol. 53, Jan. 2017. DOI: 10.1007/s11241-016-9258-z.

[23] S. Safari, M. Ansari, H. Khdr, *et al.*, "A survey of fault-tolerance techniques for embedded systems from the perspective of power, energy, and thermal issues," *IEEE Access*, vol. 10, pp. 12 229–12 251, 2022. DOI: 10.1109/ACCESS.2022.3144217.

[24] L. Abeni, R. Andreoli, H. Gustafsson, R. Mini, and T. Cucinotta, "Fault tolerance in real-time cloud computing," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, 2023, pp. 170–175. DOI: 10.1109/ISORC58943.2023.00031.

[25] K. Cao, S. Hu, Y. Shi, A. W. Colombo, S. Karnouskos, and X. Li, "A survey on edge and edge-cloud computing assisted cyber-physical systems," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 11, pp. 7806–7819, 2021. DOI: 10.1109/TII.2021.3073066.

[26] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," Aug. 2011, pp. 280–287. DOI: 10.1109/SERVICES.2011.108.

[27] F. Reghenzani, Z. Guo, and W. Fornaciari, "Software fault tolerance in real-time systems: Identifying the future research questions," *ACM Comput. Surv.*, vol. 55, no. 14s, Jul. 2023, ISSN: 0360-0300. DOI: 10.1145/3589950. [Online]. Available: https://doi.org/10.1145/3589950.

[28] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and S.-Y. Tseng, "Real-time task replication for fault tolerance in identical multiprocessor systems," in *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, 2007, pp. 249–258. DOI: 10.1109/RTAS.2007.30.

[29] Brilliant.org, "Integer equations - stars and bars," [Online]. Available: https://brilliant.org/wiki/integer-equations-star-and-bars/ (visited on 05/22/2024).

[30] R. Tarjan, "Depth-first search and linear graph algorithms," in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, 1971, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

[31] B. Peng, N. Fisher, and M. Bertogna, "Explicit preemption placement for real-time conditional code," in *2014 26th Euromicro Conference on Real-Time Systems*, 2014, pp. 177–188. DOI: 10.1109/ECRTS.2014.25.

[32] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, May 2005, ISSN: 1573-1383. DOI: 10.1007/s11241-005-0507-9. [Online]. Available: https://doi.org/10.1007/s11241-005-0507-9.