CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Balancing parallelism and time predictability for scheduling of real-time parallel tasks on multiprocessors

Master's thesis in Computer science and engineering

Amanda Paulsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Balancing parallelism and time predictability for scheduling of real-time parallel tasks on multiprocessors

Amanda Paulsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Balancing parallelism and time predictability for scheduling of real-time parallel tasks on multiprocessors
Amanda Paulsson

Supervisor: Risat Pathan
Examiner: Jan Jonsson

Master's Thesis 2023
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 2023

ABSTRACT

Balancing parallelism and time predictability for scheduling of real-time parallel tasks on multiprocessors

Amanda Paulsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As technology has advanced, real-time systems have become more common. An essential part of real-time systems is that each computational component, referred to as a task, must finish executing within a time frame, a deadline. Due to the importance of deadlines being met, scheduling tasks is crucial in real-time systems. As the demand for these systems has increased and the programs have become more advanced, a need for more computational capacity has arisen. This is because the research area of real-time systems has expanded to multiprocessor systems, with parallel execution of tasks-based applications.

One of the challenges with scheduling task sets on multiprocessor systems is to find the upper-bound on the entire execution time of the parallel application. This is a challenge because different ordering of the same set of parallel tasks, on the same system, can result in different end-to-end completion time of the application. An example of this is when a task finishes executing before its worst-case execution time, resulting in a different scheduling order of the following tasks. A reordering that leads to a longer total execution time of the program is called a timing anomaly. The state-of-the-art solutions work around timing anomalies by introducing pessimism to the upper-bound calculations, causing some task sets to be deemed unschedulable although they are schedulable. The pessimism is largely due to the fact that the internal structure of the task sets is not considered, and the dependencies between the tasks are not evaluated.

This thesis aims to eliminate timing anomalies by introducing additional dependencies, called dispatch constraints, between tasks. Such additional dependencies would remove unpredictability when scheduling and therefore also the need for pessimism when calculating the execution time. The goal is for these additional dependencies to limit the number of ways the parallel tasks can be scheduled (i.e. limit the level of parallelism), while simultaneously utilizing as much of the multiprocessor resources as possible. To find the balance between restricting the level of parallelism and utilizing the computational power a heuristic method will be used for selecting the source and destination tasks to which the dispatch constraints will be added. The heuristics developed for this are evaluated and compared with the state-of-the-art.

Acknowledgements

Firstly I would like to thank my supervisor, Risat Pathan, for his help and guidance. Risat has provided valuable insight throughout this thesis and contributed his knowledge in the area. Additionally, I want to thank my examiner, Jan Jonsson, for his input and feedback on the work. Lastly, I want to thank my opponents Frans Bergman and Shubhankar Choudhari for reviewing the work and providing comments and interesting questions.

Amanda Paulsson, Gothenburg, 2023-06-28

Contents

List of Figures

List of Algorithms

List of Tables

1

Introduction

As technology in today's society advances, more systems and machines are becoming automated. Programs and applications are no longer only expected to produce exact computations, they are now often needed to do so within a specific time frame, a deadline. The computational components, called tasks, of a system missing its deadline will lead to different consequences of varying degrees, depending on the system. These systems, called real-time systems, can be observed in for example robots used in the industry, automobiles and airplanes. An example of this is automated flight control systems in fighter aircrafts [1].

Scheduling algorithms in real-time systems are crucial in making sure that the deadlines of the tasks are being met. Real-time scheduling on multiprocessors is a big area of research. The multiprocessor hardware architecture enables multiple operations to execute in parallel. With this ability comes benefits, such as faster program execution, ability to accommodate more complex software and possibility to lower the energy consumption. An example of this is the use of multiprocessors in embedded systems, such as smartphones, where we have complex parallel applications but a limited energy source (a battery) [2]. The increased demand for multiprocessors is an effect of the market asking for more parallel processing power. However, the increased use of multiprocessors also depends on the capabilities of the existing hardware, which provides directions and limits for what is possible in terms of software development. For example, the level of parallelism when executing software depends on the number of processors in the hardware. Additionally, software common practices and industry standards are developed based on the processing capabilities of the available multiprocessor hardware.

As the field of real-time systems has evolved, an increasing demand to use multiprocessor platforms and schedule parallel tasks has occurred, and with this a new set of problems and challenges. One of the main challenges with scheduling parallel tasks on multiprocessor systems is to ensure that all the tasks will meet their deadlines at run-time. Traditionally, real-time scheduling algorithms have only considered sequential execution of tasks on single processors, but with the development of multiprocessors and their expanded possibilities, parallel execution models have become the focus [3]. In applications consisting of multiple parallel tasks, each task consists of multiple subtasks that can be executed simultaneously on different processor units. These subtasks can be scheduled in different ways depending on constraints/dependencies between each other. Different parts of an application, rep-

resented by these subtasks, often process the same data. Because of this, the order in which these subtasks are executed often (not always) matters for the result to be correct. For instance, if one subtask writes to a memory address and another subtask reads from the same memory address, then the order of these subtasks affects the outcome and correctness of the execution. There is a constraint from the first subtask to the second one because they need to be executed in that specific order, this correlation or dependency between such a pair of subtasks is called a precedence constraint.

Each subtask has an execution time. This execution time can vary, which is why something called the worst-case execution time (WCET) is used when analyzing the schedulability of tasks. The WCET is the maximum amount of time a subtask can take to execute. Since tasks meeting their deadlines is a crucial part of real-time systems, the WCET is used for determining if a task is schedulable in the system or not. When working with scheduling on a single processor, comparing the sum of the WCET for all the subtasks of a task with the task deadline is an obvious way of ensuring schedulability. If the sum is smaller than the deadline, the task is schedulable. The order in which independent subtasks execute does not matter when we consider scheduling on uniprocessor. However, on multiprocessor systems, determining the schedulability of a task is not as simple as that of on a uniprocessor. The reason for this is that since the execution time of a subtask can change at runtime the order of scheduling the subtasks on the processors can differ, resulting in different execution times for the same task when executed multiple times.

Precedence constraints between subtasks give sequential order to some parts of a task, but not the entire task the same way sequential scheduling does. Because of this, there can be more subtasks than the number of available processors 'ready' for scheduling at one time instant. The combination of multiple subtasks being schedulable at the same time with the possibility of varying execution times for the subtasks creates multiple scheduling options. This is a problem because some of these possible schedules could meet the deadline of the task, but some could miss it. Thus, the task is deemed unscheduled, even when there are schedulable options. Because of the importance of deadlines in real-time systems, the worst possible case is what determines if a task is schedulable or not.

A parallel task can be modeled or visualized as a directed acyclic graph (DAG) where the edges between the nodes in the graph represent the precedence constraint between the subtasks and the nodes in the graph represent the subtasks. When scheduling a DAG on several processors, the order of the execution of the subtasks depends on the structure of the DAG. Each node may have multiple other nodes in the graph that it can execute in parallel with. In other words, since a DAG has multiple parallel paths, some nodes may execute in parallel with other nodes. If at any time instant, there are more parallel paths than there are processors the scheduling order will depend on which one of the available subtasks is chosen (which depends on the scheduling algorithm used). As mentioned earlier, this unpredictability can result in the task being unschedulable if a schedulability analysis cannot account for such unpredictability accurately, which in turn is limiting the usage of multiprocessor systems. This limitation is sought to be avoided, especially since the unschedulable

tasks often are schedulable. There are many approaches to solving this problem, all with different success rates in terms of making previously unschedulable tasks schedulable.

This thesis aims to solve the problem of unpredictable scheduling by introducing additional constraints (called dispatch constraints) to the subtasks of a DAG. This will be done by looking at the number of parallel paths at each time instant and compare with the number of available processors. By adding a dispatch constraint from one node to one of its parallel nodes the number of parallel paths can be reduced. To find a balance between utilizing the parallelization of the multiprocessors system and keeping the task execution time predictable the goal is to introduce constraints in a way that keeps the number of parallel paths as close to the number of processors as possible. The number of parallel paths can never be larger than the number of processors to keep the scheduling predictable. In addition, when the number of possible paths is smaller than the number of processors, then we may have scenarios where there are idle processors. Unfortunately, the more we idle the processors, the longer the end-to-end execution time of the DAG tends to be. By adding dispatch constraints such that the number of possible source-to-sink distinct paths in the DAG is no more than the number of processors, the task's total execution time will be equal to the longest path in the DAG since the dispatch constraints will eliminate the unpredictability. The main challenges with this approach will be choosing the nodes to add the dispatch constraints between and adding them in such a way that the total execution time for the task will be as small as possible (to perform better than other state-of-the-art approaches).

Dispatch constraints can be added one by one until a certain condition is met. That way, limiting the number of parallel paths at all time instants can be ensured. However, since these dispatch constraints are only added based on the structure of the DAG, the effect on the schedule is still unknown. The goal is to utilize as many of the processors at all time instants as possible. The purpose is to add dispatch constraints, changing the structure of the DAG. Based on the restructured DAG the makespan can be calculated more accurately. However, there is a difference between the dispatch constraints and precedence constraints. When scheduling the precedence constraints the preceding subtask needs to finish executing before the following subtasks can be scheduled, meanwhile for the dispatch constraints the previous subtasks only have to start executing before the following ones can be scheduled. The dispatch constraints are only added for scheduling purposes, not because the subtasks data processing is related in any way (like the 'traditional' precedence constraints purpose). This means that when scheduling a DAG with dispatch constraints the added constraints can be removed if there are idle processors because of the added constraints.

The goal of this thesis is to theoretically adjust DAGs (representing tasks of applications) with dispatch constraints and also simulate the adjusted DAGs to further improve and evaluate the addition of the dispatch constraints. This is intended to improve the schedulability of tasks and task sets on multiprocessor systems. The results will be compared to the current state-of-the-art.

1.1 Contributions and organization

This thesis will introduce dispatch constraints between subtasks of tasks in multiprocessor systems. The main challenge of this thesis is the selection of the source and destination nodes the dispatch constraints will be added between. There are many possible ways of doing this and finding the optimal way would require testing all options, which is computationally not feasible. For that reason, a heuristic method will be used (more detailed explanation in later chapters). Once the DAGs with dispatch constraints have been constructed they will be simulated for further improvement and evaluation. The completed DAG execution time will be calculated and compared to the current state-of-the-art. This will be done for multiprocessor systems with one task (one DAG) using the resources.

The thesis is organized as follows. Section 2 presents a detailed description of the problem statement with background and related work. Section 3 introduces the task model and the notation used in the thesis. Section 4 presents the constructed algorithms and the heuristics. Section 5 presents the setup used to evaluate the presented algorithms. Section 6 shows the results and compares these to other methods. Conclusions, discussions, and further work is presented in section 7.

2

Background

This chapter introduces the main concepts and background for this thesis. The chapter also contains related works and description of the problem for this thesis and its focus.

2.1 Makespan

When evaluating scheduling algorithms for a parallel task something called the *makespan* is used. The makespan of a task is the worst-case completion time. If the makespan is smaller than the task deadline it confirms that the task will meet its deadline. By computing the makespan and comparing it to the deadline of the task, verification of if the timing constraints are met or not can be done. Finding the minimal possible makespan is known to be an NP-hard problem [4], and because of this, pessimism is introduced to different schedulability analyses.

Generally, there are two main ways of calculating the makespan. The first way is to find a formula that computes the makespan for an algorithm. The problem with this approach is that it often introduces pessimism, to compensate for unpredictability in the system. Because of the importance of tasks meeting their deadline, in real-time systems, the calculated makespan can never be longer than the deadline for the tasks to be schedulable. Some existing works consider the total workload and the longest path to compute the makespan. But that has the limitation that the internal structure of the DAG is not considered. Because the subtasks execution time can vary, resulting in multiple executions for the same task, using the longest path and the total execution time is not enough to compute the makespan in an accurate way. The effects, of the varying execution times, in the schedule cannot be known beforehand. Resulting in algorithms always adding extra execution time (pessimism) to the makespan, without knowing if it is actually needed for the task to meet its deadline or not. For this reason, this thesis will use the other way of computing the makespan, which is by simulating the execution. When simulating a task in a multiprocessor system the makespan will be the execution time of the processor to finish last. In order for this to be correct there can be only one way of scheduling the subtasks of a task. This thesis will achieve this by adding dispatch constraints, thus removing the unpredictability.

Figure 2.1 illustrates an example of the structure of a parallel task in the form of a DAG. The C_i value next to each of the nodes is the WCET of the corresponding

subtask. The edges in the DAG represent the precedence constraints between the subtasks. The task is executed on a two-processor system and the execution is based on a non-preemptive breadth-first search algorithm on two processors $P1$ and $P2$. The makespan of the schedule is the longest execution time of the two processors, which is 9-time units.

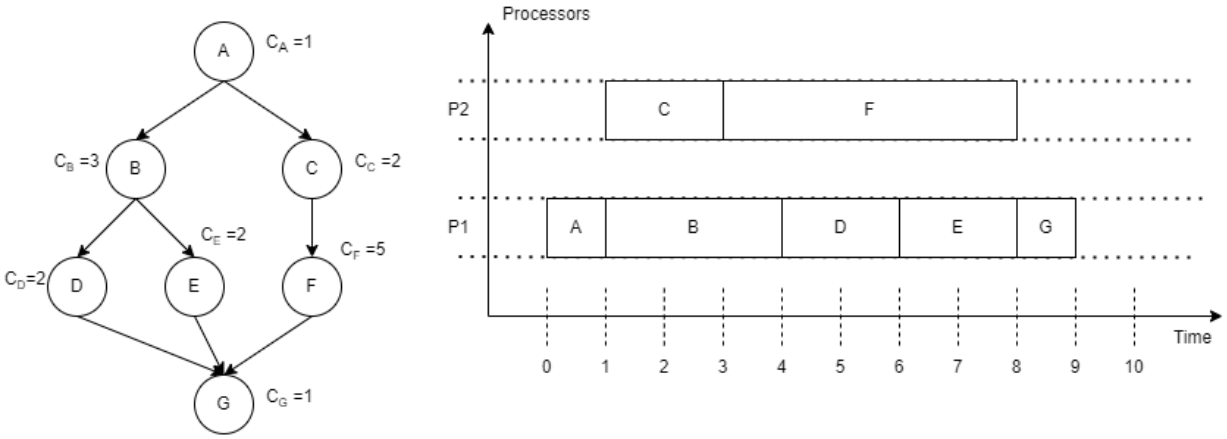


Figure 2.1: DAG representing a task and the corresponding schedule on a two-processor system.

2.2 Timing anomalies

Since some subtasks of a task have precedence constraints the scheduling possibilities of these subtasks are already determined/limited to a specific way. However, not all subtasks have preexisting precedence constraints relative to each other, which is how the unpredictability of the parallel architecture is introduced. The unpredictability means the makespan cannot be calculated easily.

One source of unpredictability is *timing anomalies*. A timing anomaly is when a change (often seen as positive) in a multiprocessor system results in an (unexpected) longer execution time. There are different types of timing anomalies. One example is an increase in number of processors. More processors are expected to lower the total execution time since it is an increase of resources, but there are scheduling cases where it results in longer execution time [5].

This thesis only considers *execution time − based timing anomalies*, which is where the makespan of a dynamically scheduled parallel application may increase when some subtasks of a task take less time than their WCET. Dynamic scheduling is when the tasks get assigned the executing cores at runtime, as opposed to static scheduling which is when tasks are pre-assigned a fixed set of cores before run-time.

Timing anomalies are a problem because it makes the system more unpredictable and therefore also makes the analysis is more difficult. An accurate analysis is critical to be able to guarantee run-time completion before the deadline.

Figure 2.2 illustrates what can happen to the total execution time when one of the subtask's execution time changes, a timing anomaly. Figure 2.2 represents the same

task as in Figure 2.1, where the task is scheduled in a two-processor system with a non-preemptive breadth-first search algorithm. The difference is that in figure 2.2 node B has a shorter execution time than in figure 2.1. Although the task in figure 2.2 has a smaller total workload (14 time units) than the task in figure 2.1 (16 time units), the schedule's total execution time is longer.

The scheduling algorithm schedules node A first. When node A has completed its execution both node B and C are released and ready for scheduling. Since both processors are idle at that time both nodes are scheduled. The next step is where the two schedules differ. This is because in the second schedule node B will complete its execution before node C, while in the first schedule node C completes its execution before node B. This difference results in nodes D and E being released and ready for scheduling before node C has finished executing in the second schedule. In the first schedule node F will be released and ready for scheduling before node B has completed its execution. In both cases, there is only one idle processor to schedule the nodes on. In figure 2.2 node D is scheduled while node E waits for an idle processor. Once node C has completed its execution node E will be scheduled on the idle processors while node F is released and scheduled after node D. Node G will be released and scheduled when all the previous nodes have finished executing. In figure 2.2 the total execution time for the task is 10 time units, which is one time unit longer than in the schedule in figure 2.1 (9 time units). Another observation between the two schedules is that the total time one of the processors is idle during the task execution is bigger in the second one (2 time units in figure 2.1 and 6 time units in figure 2.2).

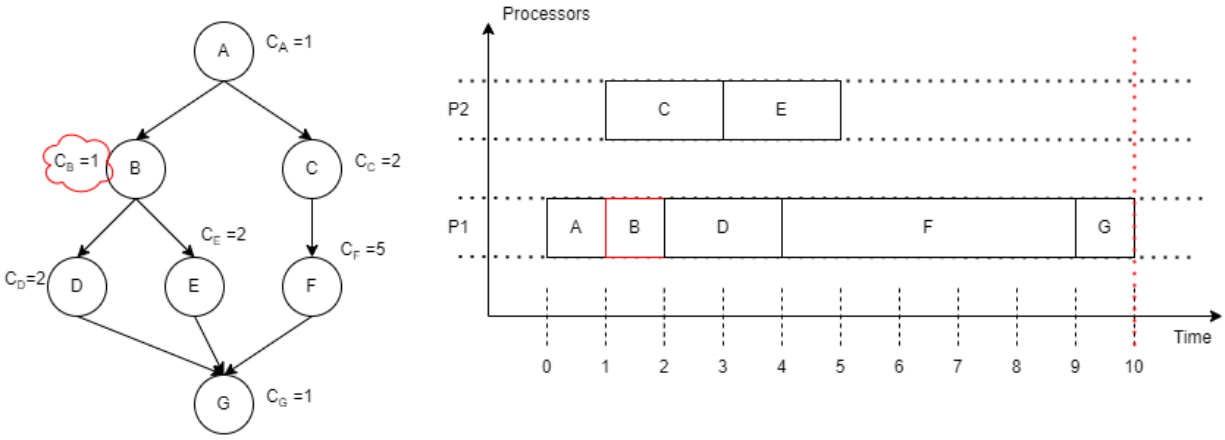


Figure 2.2: DAG representing a task and its schedule with a timing anomaly.

As the previous example show, execution time-based anomalies can happen because there are, at some time instant, more subtasks than processors ready for scheduling. Because of this, the order the nodes are released is a critical factor, which in its turn depends on the execution time of the predecessor nodes. In the DAG in figure 2.2 there are three parallel source-to-sink paths:$A-B-D-G$, $A-B-E-G$ and $A-C-F-G$. In a two-processor system, these three paths would result in multiple scheduling possibilities, which is undesired for calculating an accurate makespan without having to add extra execution time to compensate for unpredictability. For

this reason, the work of this thesis will build upon these timing anomalies to be removed, using dispatch constraints.

2.3 Dispatch Constraint

One way of avoiding timing anomalies and being able to calculate the makespan more accurately is to introduce dispatch constraints (DC) to the system. Precedence constraints are generally perceived as "restrictive" additions to the system because it prevents the full exploitation of the multiprocessor parallelism, meaning delayed execution start of some tasks. When scheduling, a precedence constraint means the source node has to finish executing before the receiver node can be dispatched. However, dispatch constraints give the system predictability because the subtasks will have a set execution order relative to each other. The source node of a dispatch constraint only needs to have been dispatched before the receiver node can be dispatched. With this, a more accurate (i.e. less pessimistic) makespan can be computed. In this thesis, dispatch constraints will be introduced to the system to eliminate the time-based timing anomalies, creating an anomaly-free schedule. Thus enabling makespan calculation by simulation: by adding dispatch constraints so that there are no more than m paths, where m is the number of processors. Then executing the task based on the modified DAG.

Figure 2.3 shows the same task represented by a DAG as in figure 2.1 and 2.2 with a shorter execution time for node B. The DAG also has a dispatch constraint added between node F and E to prevent the problem with a longer total execution time for the entire task to happen. The difference when scheduling this task is that when node B completes its execution only node D (not node E) will be released and ready for scheduling on the idle processor. After node C completes its execution node F will be ready for scheduling on that idle processor. Once Node F has been scheduled, node E (as the receiver of the DC from node F) can be scheduled as soon as there is an idle processor. Node G will be scheduled once all the previous nodes have finished their execution. Observer in this schedule, the difference between the pre-existing precedence constraints and the dispatch constraint. The node receiving the DC does not need to wait for the source of the DC to finish its execution before it can be scheduled, the source only needs to be scheduled/dispatched before. The timing anomaly is removed because there are never more subtasks ready for scheduling at one time instant than there are idle processors. There is only one option for scheduling (even when the subtask's execution time differs) and such predictability provides a tighter end-to-end completion time for the entire DAG.

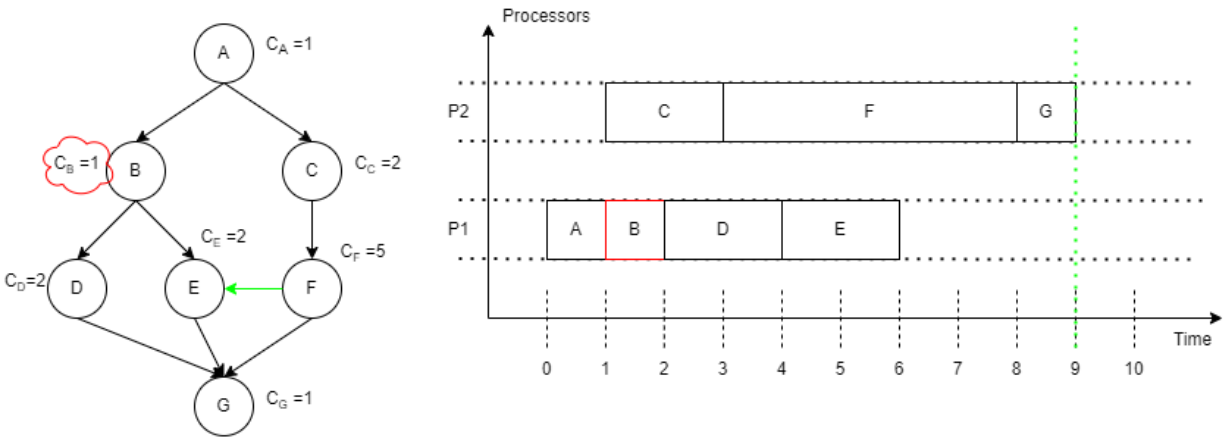


Figure 2.3: DAG and schedule representing a task with a dispatch constraint between subtask F and E.

2.4 Processor Scheduling

There are two main ways of scheduling tasks on processors; *global scheduling* and *partitioned scheduling*. Global scheduling is when the tasks can be assigned to any processor dynamically. Partitioned scheduling is when the tasks are statically assigned to a processor and migration between processors is not permitted.

However, *federated scheduling* is another approach and is a generalization of partitioned scheduling for parallel tasks on multiprocessor systems. Using federated scheduling each parallel task will be assigned a subset of the processors from a particular multiprocessor platform. In federated scheduling, tasks are divided into two sets, one containing all high-utilization tasks and one containing all low-utilization tasks. Federated scheduling algorithms then divide the system resourced between these two task sets depending on the implementation. One example is that the algorithm first assigns cores to the high-utilization tasks based on the execution time, worst-case critical-path length and the deadline. The low-utilization tasks, which can run sequentially, get assigned to the remaining available cores [6].

2.5 Related Work

Because of the potential and the increasing use of multiprocessor systems, there is research about scheduling algorithms for tasks on multiprocessors. With this also many different approaches with different parts of the systems in focus. Most of the focus has previously been on sequential scheduling, where the problem is to schedule many sequential tasks on multiple cores. Since parallel tasks introduce additional complexity one of the biggest challenges today is to maximize performance while also fulfilling the real-time constraints of parallel multiprocessor systems.

One of the first works on scheduling on multiprocessors was done by Graham in [5]. Graham describes timing anomalies in multiprocessor systems and also defines

different kinds of timing anomalies. The work gives some exact bounds on the extent of how these anomalies can affect the execution time for dynamically scheduled task sets.

With the increase of research in the area different real-time task models have been proposed in attempts to express the parallelism of applications. Some examples are; the *fork-join* model [7], the *synchronous parallel* task model [8] and the *sporadic graph model* [9]. All of these models divide the tasks into smaller subtasks that are allowed to run in parallel. The fork-join model sequentially executes a task until it encounters a parallel sequence, at which it "forks" into multiple parallel executions. After the parallel execution the task "joins" back to sequential execution. The parallelism is usually fixed and limited to the number of processors available. The synchronous parallel task model is an extension of the fork-join model where parallel execution can be done successive and the degree of parallelism is not limited. Both the fork-join and the synchronous parallel models have dependencies between the segments of nodes of the task, a node has to wait for all the other nodes of the previous segment to finish executing before it can begin executing. The DAG model is more general than both these models because the execution of each node only depends on the precedence constraint. Tasks in this model are represented by a directed acyclic graph where directed edges between the nodes represent the precedence constraints between the code segments. In this thesis, the DAG model is used because the structure of the DAG (dependencies between the subtasks of a task) will be used to remove timing anomalies.

2.5.1 Multi-DAG scheduling

One of the first attempts at a parallel model that takes control-flow information and conditional statements into consideration is by Saifullah et al [10]. They present a multi-DAG model where each task is represented by a set of execution paths throughout the task code and is modeled as a DAG of sub-tasks. The motivation behind this work is that previous parallel task models either, assume that all threads of a parallel task must execute every time (which never happens with conditional statements), or the model always takes the worst-case option. The model [10] works in two steps. The first step is creating separate graphs for the different execution flows (different possible conditions). That way during run-time, when the task takes one of the execution flows, the required budget to finish all of the sub-tasks is known. In this first step, the constructed graphs are organized as sets of segments, where directed edges only exist between the adjacent segments (every node within a segment is connected to every node in the next segment). The second step is then to merge the DAGs created into a single synchronous parallel task that still fulfills the precedence constraints. This multi-DAG model improves on schedulability tests for DAGs where the internal structures are considered. Like the multi-DAG model, this thesis will consider the internal structure of the DAG and the varying execution times of the subtasks.

2.5.2 Conditional parallel task scheduling

The DAG model has also been extended to conditional constraints by Melani et al in [11]. This was done to provide a tighter analysis (makespan) of parallel task systems. This model is called the *conditional parallel* task (cp-task) model. Each DAG represents a task that can contain both parallel and conditional nodes. The schedulability analysis is based on the concept of interference, which is extended to define the concepts of critical chain and critical interference. Interference is the sum of time when a task is ready but cannot execute because there are no idle cores available. The critical chain of a cp-task is defined as the chain of nodes that leads to the worst-case response time. Critical interference is defined as the total time in which any critical node of a critical chain is ready but cannot execute because there are no idle cores. The interference calculation is divided into two different parts; inter-task interference and intra-task interference. Inter-task interference refers to the amount of time a task is ready for scheduling but can't be scheduled because other tasks are occupying the processors. Intra-task interference refers to the amount of time that a subtask is ready for scheduling but can't be scheduled because other subtasks of the same task are executing. Interference is used to enable the calculation of an upper-bound of the worst-case response-time of each cp-task. In this thesis, the first step will be to consider single-task systems (single DAGs), when doing so the internal structure and therefore also the intra-task interference will be considered. The extension of this will then be to consider multi-task systems, where both inter-task and intra-task interference will be considered.

To check the schedulability of a conditional DAG task system, by calculating the upper-bound of the worst-case response-time, two parameters for each cp-task are needed; the worst-case workload (WCW), and the length of the longest chain in the DAG. These two parameters are together with the number of processors used to calculate the makespan. The longest path of a cp-task can be calculated the same way as in a non-conditional DAG because the conditional nodes dont affect the outcome. This can be done in linear time relative to the size of the DAG by using standard techniques. However, the worst-case workload is more complicated to calculate since the conditional nodes need to be considered. The paper presents an algorithm for doing this, in quadratic time relative to the size of the DAG. Furthermore, the algorithm is extended to disregard the cases where the total WCW and the critical path represent different paths in the DAG since this can never realistically happen simultaneously. These cases introduce pessimism to the calculations and should therefore be ignored. The paper validates the algorithms/results by simulations for both global fixed-priority and global earliest deadline first scheduling algorithms.

2.5.3 The LAZY scheduling

The work [12] provides a proven execution time-based timing anomaly-free scheduling algorithm, called the *lazy scheduler* (LAZY). The lazy algorithm is a priority-based, non-greedy, and non-preemptive algorithm. It consists of two main parts; priority assignment and the lazy scheduler. The priority assignment policy works by assigning each node in the DAG a fixed priority. This is done based on the priority

of the parent nodes. Parallel tasks generated by the same parent usually need to synchronize their results. Therefore synchronization nodes are used. Since these synchronization nodes can become bottlenecks in the scheduling they are assigned the highest priority amongst the nodes on the same level, that way, more of the parallelization benefits can be utilized. Because the priority for each node is only based on the parent node this can be done at runtime, no global information is needed. The second step, the lazy scheduling, works by scheduling the highest-priority ready tasks that are dispatched on the idle processors. Because of the subtasks fixed priority timing anomalies are avoided, even when some subtasks finish executing before the WCET. This gives the scheduling predictability the same way the work in this thesis intends to do. The makespan for the LAZY scheduler is determined by simulation, which will give a safe estimate because the timing anomalies have been eliminated. Since this thesis also will eliminate timing anomalies simulation can be used for determining the makespan.

2.6 The problem

The objective of this thesis is to devise an efficient makespan computation technique. Most previous work for this has been done through mathematical formulations. These mathematical formulations rarely consider the structure of the tasks and therefore have to introduce pessimism to the upper bounds of the calculations of makespan. To avoid adding pessimism to the makespan, with the hope of improving the makespan calculations, this thesis will compute the makespan based on the simulation of the multiprocessor scheduling. To do this a scheduling algorithm will be assumed to exist and an already existing DAG will be assumed.

An offline technique will be applied to the given DAG to transform it into a DAG with dispatch constraints. The transformed DAG will be used as input to the scheduling algorithm. The scheduling algorithm will schedule the nodes of the DAG on a multiprocessor system. The total time of the produced schedule will be the makespan (a proof sketch to formally prove this will be given later).

Using simulation to calculate the makespan of tasks is a small area of research where not a lot of work has been done so far. One of the examples of work that have used simulation is the LAZY scheduling in [12], in which priorities are assigned to all nodes. Assigning priorities to all nodes can be problematic if there are a lot of nodes, which is why this thesis will use dispatch constraints instead.

The source and receiver node for each dispatch constraint need to be selected. There are many possibilities for this selection. Finding the best option would require an exhaustive search which is computationally very heavy and therefore unfeasible. Because of this, the main challenge of the work is to come up with effective heuristics so that the source and receiver nodes for the dispatch constraints can be selected in the best way possible.

The heuristic technique is an approach to solving a problem in a way that is not guaranteed to be optimal, sufficient, or rational but provides a result in (often) shorter time. The optimal solution would be picking the source and destination

nodes, for the dispatch constraint, that gives the best result. The best result is when the makespan is as small as possible because all the processors are being utilized. A very simple example of a dispatch constraint that could be introduced to a system is that if one node has an execution time that is 10% of the whole DAG (total workload), then there could be a dispatch constraint from that node to all the other ones. This node would then be dispatched and executed early in the schedule because all the other nodes cannot be scheduled before that node. This ensures that the nodes with the longest execution time will be scheduled early. That could prevent one processor from executing longer than the others in a multiprocessor system, which could minimize the total execution time of the DAG.

This thesis will use the work by Melani et al in [11] as a baseline (explained in section 2.5). The limitation of the work done by Melani et al is that only two parameters are used to characterize the whole DAG, the worst-case workload (W) and the longest path (L). This is limiting because the internal structure of the DAG is not considered. Not considering the internal structure is compensated for with a more pessimistic computation of the makespan, which can result in tasks being deemed unscheduled when they are schedulable. This work aims to exploit the internal structure of the DAG so that dispatch constraints can be added. The resulting more structured DAG will be exploited to calculate the makespan in a much less pessimistic way. The next paragraph presents an example that demonstrates how the makespan is computed in the state-of-the-art (Melani et al [11]). The example also shows how by adding dispatch constraints the results can be improved.

Example: Figure 2.4a illustrates a simple task DAG, where C_i next to the nodes denotes the workload for the subtasks. Assuming a 2 processor system and the same scheduling algorithm as in section 2.1. Based on the work in [11] the system's longest path is $L = 12$ (node path: $A - D - F$), and the worst-case workload is $W = 17$. Using the algorithm by Melani et al. presented in equation 2.1. The upper-bound of the worst-case response time (Z) is calculated using these two parameters. The result for two processor systems with the DAG from figure 2.4a will result in a makespan of 14.5 time units ($12 + \frac{1}{m}(17 - 12) = 14.5$)

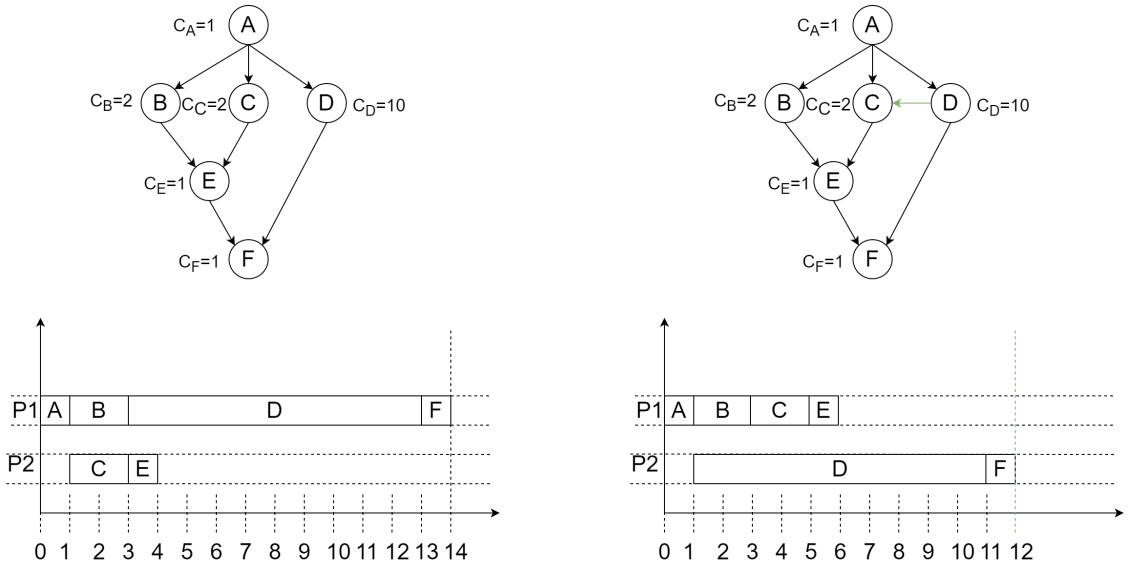
$$Z = L + \frac{1}{m}(W - L) \tag{2.1}$$

14.5 is a safely calculated makespan, which is a bit pessimistic compared to the dispatch constraint-based analysis.

By introducing dispatch constraints, the parallel paths in the system can be reduced. From three to two parallel nodes at each time instant. In figure 2.4a the three nodes are released at the same time: B, C and D. Since it is a two processor system only two of them can be dispatched at that time. By adding dispatch constraints to the DAG the number of parallel paths will be reduced and thus also the makespan. A dispatch constraint between node D and C in the system means node C will not be put in the ready queue until node D is dispatched, see figure 2.4b. Note in the figure, that there is a difference between the 'traditional' precedence constraints and the dispatch constraint. Node C (with a dispatch constraint from node D) can be scheduled once node D has been dispatched and there is an idle processor

available. If the dispatch constraint would be a 'traditional' precedence constraint node C wouldn't be schedulable until after node D finished executing. The dispatch constraints affect the scheduling algorithm not the structure of the task.

Adding a dispatch constraint reduces the makespan of the system to 12 time units because it is no longer possible for nodes both B and C to be scheduled at the same time as node D (there are no longer multiple scheduling possibilities). The added dispatch constraint ensures that the node with the largest execution time, is not being interfered with because it will be dispatched as soon as it is ready. The problem to solve is *how* to introduce the dispatch constraint and to prove that there are no timing anomalies if some node takes less than its worst-case execution time.



(a) Task represented by a DAG and its schedule.

(b) Same task but with an added dispatch constraint, and its schedule.

Figure 2.4: A task represented by two DAGs, one with introduced dispatch constraint. The schedules for both DAGs showcase different execution times for the same task.

3

System Model

This chapter defines the real-time multiprocessor task model and describes the scheduling of the tasks.

3.1 Task Model

This thesis considers homogeneous multiprocessor platforms with m identical processors of the same speed. In the area of real-time systems each individually executing part of a program is called a *task* (τ_i). Each task has a worst-case execution time (WCET). All tasks also have a *relative deadline* which indicated, relative to the arrival of the task, when it latest can finish. Each parallel task consists of instances called *subtasks* and can be modeled with a *directed acyclic graph* (DAG) $G = (V, E)$. A sporadic DAG task set with n tasks is defined as $\tau = \{\tau_1, ..., \tau_n\}$, where each τ_i is defined as one DAG with the index $i \in \{1, 2, ..., n\}$. Each task consists of k subtasks. In a DAG each node $v_j \in V_i$ represents a subtask, where the index $j \in \{1, 2, ..., k\}$. In this thesis, the terms "subtask" and "node" are used interchangeably. Each directed edge represents a precedence constraint between the nodes. A node is *ready* to execute when all of its predecessors have finished executing. If (v_1, v_2) is a directed edge in the DAG then v_1 must complete execution before v_2 can begin executing. Each node of a task τ_i is separated by a period T_i and has a relative deadline $D_i \leq T_i$. The total workload of a task W_i is the sum of the subtasks workload C_v. The utilization of a task U_i is the task workload W_i divided by the task period T_i, $U_i = W_i/T_i$.

A task is said to be *periodic* if the arrival time between the subtasks is constant. The time between the arrival of the subtasks is then called a period. Not all tasks are periodic. When the subtasks do not arrive strictly periodically, they may arrive after the period, then it is called a *sporadic* task.

3.2 Scheduling Algorithm

This section describes the scheduling algorithm used for tasks where dispatch constraints have been introduced and how the scheduling differs from tasks with only precedence constraints. The dispatch constraint scheduler being timing anomaly free is a conjecture. The proof of this is left as future work.

3.2.1 The dispatch constraint scheduler

This dispatch constraint scheduling algorithm is non-preemptive, meaning once a subtask is assigned to a processor it completes its execution on that processor. During scheduling, ready subtasks from the ready queue are dispatched to idle processors if two conditions are met. Before presenting these conditions the following notation and definitions are needed.

Each node (v_j) of a task (τ_i) has a set of predecessor nodes (P_j). The nodes in the P_j set all have precedence constraints to the node v_j. Each node also has a set of nodes (DP_j), containing the predecessors that have dispatch constraints to the node. The P_j set and the DP_j set cannot overlap for one node. A node is called *active* at time t if it has been released but not finished executing. For a node to be considered active both the P_j set and the DP_j set must be empty. A subset of the active nodes is the ones that have been dispatched but not finished execution at time t and is denoted by DA_t. The nodes in the ready queue are active but have not been dispatched yet, denoted by RQ_t. The set of $DA_t \cup RQ_t$ contains all the active nodes at time t.

Assuming the system starts at time $t = 0$, all processors are idle and RQ_0 only contains the root of the task DAG. The scheduler makes a new scheduling decision at time t when the following scheduling event **SE** occurs:

- **Scheduling Event SE:** when $t = 0$ or some task completes its execution while not all nodes in the system have been dispatched.

The scheduling algorithm stops when, at time t, the SE event occurs but all processors are idle and all nodes have finished executing. When an event SE happens at least one processor will be idle and a new node needs to be selected to be dispatched. For a node to be put in the ready queue and therefore available for scheduling both the following conditions are required to be met:

- All nodes in the P_j set must have completed executing.
- All the nodes in the DP_j set must have completed executing or be dispatched.

A more detailed description of the scheduling used for tasks with dispatch constraints is presented in Algorithm 1.

The scheduler takes the task DAG and the number of processors m as input. It dispatched the nodes of the DAG to idle processors based on the sets of predecessors P_j and predecessors with dispatch constraints DP_j for each node. The active nodes, that have not been dispatched, are put in the RQ (line 3). Once a node is active (in the ready queue), which means all its predecessors have completed their execution and it does not have any dispatched constraints, the node gets scheduled when a processor becomes idle, lines 4-5. After a node has been dispatched it is removed from all the other nodes' sets of predecessors P_j and dispatched predecessors DP_j, line 6. Line 4-6 in Algorithm 1 is repeated until the ready queue is empty. For the cases when the ready queue is empty but not all nodes have been dispatched yet and there are idle processors lines 8-11 are performed. What happens then is that all other nodes in the DAG are checked, one at a time based on index order. If a

Algorithm 1 The dispatch constraint scheduler

```
1: procedure SCHEDULER(DAG, m)
2:     if SE then
3:         RQ ← active nodes
4:         while there is an idle processor & nodes yet to be dispatched do
5:             if RQ is not empty then
6:                 Dispatch node x from RQ
7:                 for All nodes y in DAG do
8:                     if DP_y contains x then
9:                         Remove x from DP_y
10:                    end if
11:                end for
12:            else
13:                for Every node z not dispatched do
14:                    if isempty(P_z) and all nodes in DP_z are dispatched then
15:                        Dispatch node z
16:                        for All nodes y in DAG do
17:                            Remove z from DP_y
```

node has an empty P_j set, the DP_j set is checked (if both the P set and the DP set are empty the node will have already been put in the RQ). If all the nodes in the DP_j set are dispatched or have finished executing, the node can be dispatched as well. The steps in lines 8-11 are repeated until there are no idle processors left at time t or there are no nodes that meet the conditions.

In this thesis the dispatch constraint scheduler (algorithm 1) will be used to compute the makespan, by taking a dispatch constraint DAG as input and scheduling it. The total execution of the DAG will be the makespan of the task.

Formal proof of the dispatch constraint scheduler being timing anomaly free is left as future work. The schedule can be ensured to be timing anomaly free by adding additional dispatch constraints to the simulated schedule. A set of dispatch constraints added to the original DAG is used to, by simulation, calculate the makespan of the DAG on a multiprocessor platform. However, during simulation some nodes may be released earlier than in the original dispatch constraints DAG. This out-of-order execution can be captured by adding additional dispatch constraints after the simulation is over. By adding these additional constraints the subtasks are guaranteed to execute in this particular order, and is therefore timing anomaly free.

3.2.2 The classic scheduler

The state-of-the-art scheduling algorithm, the dispatch constraint scheduler in section 3.2.1 is based upon, is a non-preemptive breadth-first-search algorithm.

Algorithm 2 The classic scheduler

```
1: procedure SCHEDULER(DAG, m)
2:     if SE then
3:         RQ ← active nodes
4:         while idle processor & nodes yet to be dispatched do
5:             if RQ is not empty then
6:                 Dispatch node from RQ
7:                 Remove from all other nodes P
```

Just like the dispatch constraint scheduler in section 3.2.1, the classic scheduler described in algorithm 2 also takes a DAG and the number of processors m as input. This DAG is unmodified, without the added dispatch constraints. This means the nodes do not have DP sets. When an event SE happens at time t all the active nodes are put in the RQ_t set, line 3. If there are any idle processors the nodes get dispatched from the RQ_t set, in the same order they were added, which is the increasing index (lines 5-6). Once a node gets dispatched it is removed from all other nodes P lists (line 7).

A note on real implementation: An example of the implementation of precedence constraints in real systems is OpenMP's task dependencies. OpenMP is a parallel programming model based on the fork-join model. Task dependency is a part of OpenMP's tasking concept, which enables the parallelization of applications. Parts of the application's code are defined as different tasks that can execute in parallel. The task dependency enables you to introduce additional constraints on the scheduling between tasks. The tasks can for example be while-loops or recursive functionality. When it comes to dispatch constraints in OpenMP the priority clause can be used. The priority clause specifies the task execution order by assigning priority values to the tasks. Among all tasks ready to be dispatched the ones with the highest priority value will be prioritized. Source nodes of the dispatch constraints having higher priority than the receiver nodes mean they will be scheduled first, thus creating an executing order and relation between the nodes. Exploiting different options for real implementation is left as future work.

4

Constructing Dispatch Constrained DAG

The purpose of this thesis is to explore and analyze the data structure of DAGs representing tasks. This is to add additional internal constraints between nodes in the DAGs. These constraints are dispatch constraints with properties intended to improve the scheduling. The main objective is to find the source and destination nodes to add dispatch constraints between them. To do this effectively an algorithm has been constructed that is presented later in this chapter.

The proposed algorithm will take the original DAG and return the modified DAG with the dispatched constraints added. The modified DAG will have the number of parallel paths minimized so that it is never more than the number of processors. Minimizing the number of parallel paths ensures that, at runtime, there are no more nodes ready for execution than there are available processors. This means that unpredictability in the form of timing anomalies has been eliminated from the schedule. The ready queue will not have nodes waiting. As a part of the proposed algorithm, different heuristics have been included to add dispatch constraint to the DAG (the heuristics are explained in chapter 2.6). Two heuristics are considered, one for selecting the source node and one for selecting the destination node for the dispatch constraints.

The proposed algorithm has been constructed to limit the number of parallel paths for each task. To eliminate timing anomalies the number of parallel paths at each time instant cannot be more than the number of processors in the system. This is to eliminate the possibility of multiple different schedules. As mentioned in previous sections this will eliminate pessimism introduced in the algorithm presented by Melani et al in [11]. Because of this approach, the makespan of the tasks will be equal to the longest path in the DAG, even when not using the scheduling algorithm to calculate the makespan(presented in the previous chapter). When using the algorithm the makespan can be made smaller.

4.1 The construction of DC DAG

This section describes the first version of an algorithm to find a dispatch constraint DAG.

Algorithm 3 Construct a DAG with dispatch constraints

```
1: function DC_DAG(V,m)    ▷ Where, V=DAG and m=number of processors
2:     TANodes = empty
3:     for v_i ∈ V do
4:         ownPath_i ← SamePath(V, i)
5:         parallelNodes_i ← NotSamePath(V, i)
6:         if length(parallelNodes_i) + 1 > m then
7:             TANodes = TANodes ∪ {v_i}
8:         end if
9:     end for
10:    if isempty(TANodes) then
11:        return V
12:    else
13:        vS ← pickSourceNode(TANodes, ownPath, parallelNodes, V)
14:        vR ← pickReceiverNode(vS, ownPath, parallelNodes, V)
15:        V ← edge from vS to vR
16:        DC_DAG(V,m)
17:    end if
```

The first version of the solution, presented in Algorithm 3, takes a DAG and the number of processors m as input and produces an updated DAG with a limited number of parallel paths as output. The algorithm works by first, for each node (v_i) in the DAG, dividing the other nodes in the DAG into either the *ownPath* or the *parallelNodes* set (line 3-5). The *ownPath* set represents all the nodes that cannot execute in parallel with v_i, meaning all such nodes are in successor paths and predecessor paths of v_i. This is done using the function SamePath(V, i) presented in Algorithm 4 and the function NotSamePath(V, i) presented in Algorithm 7. The *parallelNodes* set contains all the nodes that can execute in parallel with v_i, all nodes in the DAG that is not a part of either the successor paths or predecessor paths. Then, at line 6, the next step is checking the size of the *parallelNodes* set for all nodes. This is because, if *the size of the group* +1 (add one to include the node itself) is bigger than m, the number of parallel nodes is more than the available processors.

When the number of parallel nodes is greater than m, timing anomalies can occur, which means the longest path of the schedule cannot be used as the makespan (meaning the approach in this thesis would not work). When the number of parallel nodes is smaller than m, then for the worst case the entire task would be scheduled sequentially. Sequential scheduling happens if we add dispatch constraints without any consideration to the level of parallelism we are reducing, which would result in the biggest possible makespan (the total workload of the DAG). Since the goal is to improve over the state-of-the-art approaches, sequential scheduling is not desirable. The ideal case is for the number of parallel nodes to be exactly equal to m because then all processors are utilized, which would result in a smaller makespan. Because of this, all nodes with more than $m + 1$ parallel nodes are put in the $TANodes$ set, line 7. If the $TANodes$ set is empty the algorithm returns the DAG (V), no more

dispatch constraints need to be added (lines 10-11).

At line 13 the source node for the dispatch constraint is selected. This is done with the function pickSourceNode($TANodes, ownPath, parallelNodes, V$). There are many ways of choosing this node. The goal of the algorithm is to get the number of parallel paths as close to the number of processors in the system as possible while also keeping the total execution time (longest path) of the system as small as possible. Because of this, there is no way of knowing what the optimal nodes to select for adding dispatch constraints are. The *pickSourceNode* function represents the first heuristic part of the algorithm, the different heuristics tested are presented in section 4.3. The input for this is $TANodes$, $ownPath$, $parallelNodes$ and the DAG V. The $TANodes$ set is the set of nodes with more parallel nodes than there are processors, which is the group of nodes the source node will be selected from. The sets $ownPath$ and $parallelNodes$, for all the nodes of the DAG, and the DAG itself V is used because the node will be picked based on the structure of the DAG.

When the source node for the dispatch constraint is chosen the next step, line 14, is to select one of its parallel nodes as the receiver of the dispatch constraint. This is where the second part of the heuristic method comes in because also this node can be chosen in many different ways. The different heuristics for this are presented in section 4.4.1. This function takes the already selected source nodes vS as input, this is to know what node will be the parent of the receiver node. It also takes the DAG and, the sets of $ownPath$ and $parallelPaths$ to use the structure when selecting the receiver. On line 15 the edge between vS and vR is added to the DAG.

When the receiver has been selected and the dispatch constraint has been added to the DAG the algorithm runs recursively, line 16. The calculations of $ownPath$ and $parallelNodes$ require to be done recursively because adding a constraint can affect the structure of the entire DAG. This means that the $parallelNodes$ and $ownPath$ set may change for all nodes. To understand the total effect of the added dispatch constraint and to determine if more dispatch constraints need to be added, all the earlier steps require to be recalculated. The function returns the modified DAG, at line 11, when there are no longer any nodes with a set of parallel nodes bigger than the number of processors. The possibility of any timing anomaly has been removed and the modified DAG can be scheduled without unpredictability.

Algorithm 4 All nodes in the same paths as node i

```
1: function SamePath(V,i) ▷ where V is the DAG and i is the index of the node
2:     nodesInPath = empty
3:     nodesInPath = nodesInPath ∪ PredecessorPaths(V, i)
4:     nodesInPath = nodesInPath ∪ SuccessorPaths(V, i)
5:     return nodesInPath
```

Algorithm 4 (SamePath(V, i)) takes the DAG (V) and the index (i) of a node as input. The function returns the set of all nodes that are in the same paths as node i in the DAG (line 5). This is done by calling two other functions, one that returns the set of all the predecessor nodes (PredecessorPaths(V, i)) and

one that returns the set of all successor nodes (SuccessorPaths(V, i)), line 3-4 (presented in Algorithm 5 and 6).

Algorithm 5 All nodes in the predecessor paths of node i

```
1:  function PredecessorPaths(V,i)   ▷ where V is the DAG and i is the index of the node
2:      allPredecessors = empty
3:      if pred(i) = NULL then
4:          return allPredecessors
5:      end if
6:      for x = nodes in pred(i) do
7:          allPredecessors = allPredecessors ∪ x
8:          allPredecessors =allPredecessors ∪ PredecessorPaths(V, x)
9:      end for
10:     return allPredecessors
```

Algorithm 5 (PredecessorPaths(V, i)) takes the DAG (V) and the index (i) of a node as input and returns a set of all nodes in predecessor paths to node i. Each node of the DAG has its own set containing its predecessors ($pred(i)$), all the nodes with precedence constraints to node v_i. Each of the node's predecessors (line 6) in this set, is put in the $allPredecessors$ set (line 7). The function is then run recursively (line 8) for all the predecessor nodes. That way all of the nodes from node i to the root of the DAG will be added to the $allPredecessor$ set. The function returns when it gets to a node with no predecessors (which is the root), lines 3-4.

Algorithm 6 All nodes in the successor paths of node i

```
1:  function SuccessorPaths(V,i)   ▷ where V is the DAG and i is the index of the node
2:      allSuccessors = empty
3:      if succ(i) = NULL then
4:          return allSuccessors
5:      end if
6:      for x = nodes in succ(i) do
7:          allSuccessors = allSuccessors ∪ x
8:          allSuccessors = allSuccessors ∪ SuccessorPaths(V, x)
9:      end for
10:     return allSuccessors
```

Algorithm 6 (SuccessorPaths(V, i)) takes the DAG (V) and the index (i) of a node as input and returns a set of all nodes in successor paths to node v_i. The function is structured in the same manner as Algorithm 5 but instead of the set of predecessors $pred(i)$, each node also has a set of successors $succ(i)$ that is used. Each of the nodes with precedence constraint from node i is put in the $allSuccessors$ set, line 7. The function is then run recursively (line 8) for each of these nodes. The

algorithm returns the set of all nodes from node v_i to the sink node of the DAG (at line 10).

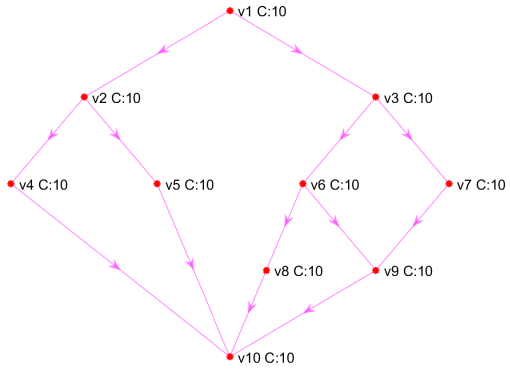
Algorithm 7 All nodes in the DAG that are not in any of the same paths as node i

```
1: function NotSamePath(V,i) ▷ where V is the DAG and i is the index of the node
2:     nodesInPath= empty
3:     nodesInPath = nodesInPath ∪ SamePath(V, i)
4:     nodesInPath = nodesInPath ∪ i
5:     return allNodes - nodesInPath
```

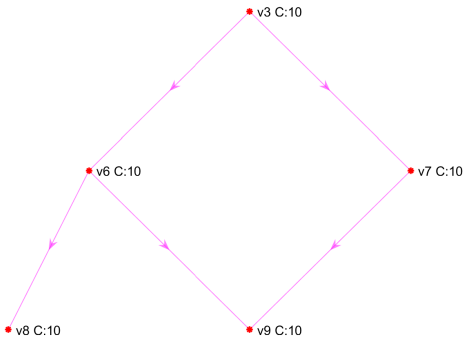
Algorithm 7 takes the DAG (V) and the index (i) of a node as input, and returns the set of nodes that can execute in parallel with node v_i. This is done by calling the SamePath(V, i) function (line 3), which returns the nodes that share a path with v_i (presented in Algorithm 4). The node itself is then also added to the same set $nodeInPath$ on line 4. This is because the node can't execute in parallel with itself. All nodes in the $nodesInPath$ set are then subtracted from the set of all nodes in the DAG. The remaining nodes will be the nodes that can execute in parallel with node i, which is returned at line 5.

One limitation of the algorithm 3 (DC_DAG) is that it adds pessimism to the result by only considering the number of parallel nodes, which ignores the internal structure of the DAG. Figure 4.1b shows the parallel nodes (the subgraph), that can execute at the same time as node $v2$ in figure 4.1a. There are five nodes in the subgraph, assuming $m = 4$, the DC_DAG algorithm would result in (at least) one added edge from $v2$ to one of these parallel nodes. Figure 4.1c shows the result of running the DC_DAG algorithm with the DAG in Figure 4.1a as input (edges between the node $v2 \rightarrow v3$ and $v4 \rightarrow v3$ have been added).

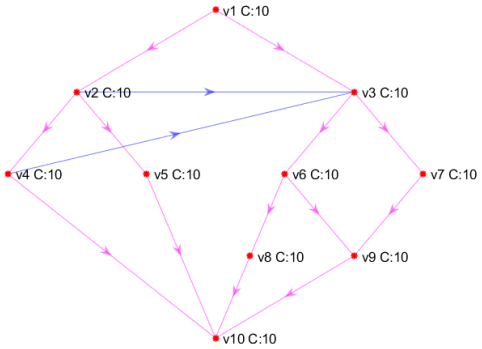
The method for selecting the source node for the added constraints in this algorithm is presented in 4.3.1 and is based on the node with the biggest workload. The method for selecting the receiver node for the added constraints is presented in 4.4.1 and is based on selecting the node with the smallest workload. By looking at the figures the observation can be made that all nodes have the same workload. Because of this, the source and receiver nodes are chosen based on the order they are added to the lists, which is the increasing order of the index. The structure of the DAG did not contribute to the selection process. Counting the paths in figure 4.1b shows there are three possible paths ($v3 - v6 - v8$, $v3 - v6 - v9$, $v3 - v7 - v9$), meaning there are only three paths that can execute at the same time as node $v2$. Therefore, no extra edge needs to be added from node $v2$ to one of the parallel nodes. Based on this observation section 4.2 proposes our refined algorithm.



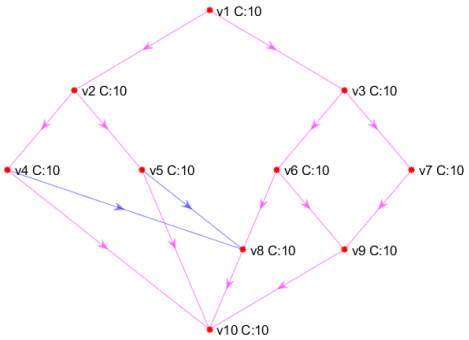
(a) Task represented by a DAG.



(b) The parallel nodes of node $v2$.



(c) DAG with added dispatch constraints between nodes $v2 \rightarrow v3$ and $v4 \rightarrow v3$ (added with algorithm *DC_DAG*). The longest path is 70.



(d) DAG with added dispatch constraints between nodes $v4 \rightarrow v8$ and $v5 \rightarrow v8$ (added with algorithm R_DC_DAG). The longest path is 50.

Figure 4.1: Task represented by a DAG, the subgraph representing the parallel nodes of node $v2$, and the modified DAGs for algorithms *DC_DAG* and *R_DC_DAG*.

4.2 Refined construction of DC DAG

Because the first version (*DC_DAG*) does not consider the internal structure it was further refined to consider the parallel paths instead of the parallel nodes in the $TANodes$ set. This second version of the algorithm (*R_DC_DAG*) takes the parallel nodes (the subgraph of the DAG) and counts the number of possible paths (line 5), which often is smaller than the number of parallel nodes. For example figure 4.1b shows five parallel nodes of node $v2$ but only three parallel paths. The algorithm used for computing the number of paths is presented in Algorithm 9. This is done for all nodes in the DAG V. If there are nodes in the DAG that have more than $m - 1$ *parallel paths* a dispatch constraint needs to be added. The source for the constraint is selected with the same method as for the first version of the algorithm (further described in section 4.3). The receiver node is selected differently compared to *DC_DAG*, at line 11. This is because the goal is to get the total

number of parallel paths as close to m as possible to utilize as many of the available processors. The *minimumPathReduction* function used is presented in Algorithm 13 (section 4.4.2). This algorithm iterates over all of the nodes in the subgraph (for the selected source node) and temporarily adds a dispatch constraint to all of them, one at a time. For each temporarily added dispatch constraint the number of possible parallel paths is recalculated. The recalculations are then used to select the node that adds a constraint to give the smallest reduction of the number of paths. The reason for keeping the reduction of number of paths as small as possible is to prevent the introduction of too much pessimism (less sequential execution), by making processors idle.

An added dispatch constraint will lower the number of parallel nodes for the source node by at least one. Just like in *DC_DAG* the effects of the added constraint, on the structure of the DAG, can not be known without recalculation of the sets *ownPath* and *SamePath* for each node. The refined algorithm, called *R_DC_DAG*, will therefore run recursively until the number of parallel paths is smaller or equal to the number of processors (line 8), for all time instants. Just like in algorithm *DC_DAG*, the *R_DC_DAG* returns the modified DAG, at line 9, when the $TANodes$ set is empty (no node has more than $m - 1$ parallel nodes). There are then no nodes in the DAG with more parallel paths than the number of processors in the system.

Algorithm 8 Refined construction of DAG with dispatch constraints

```
1:  function R_DC_DAG(V,m)          ▷ Where, V=DAG and m=number of
    processors
2:      for v_i ∈ V do
3:          ownPath_i ← SamePath(V, i)
4:          parallelNodes_i ← NotSamePath(V, i)
5:          TANodes ← computeNumPaths(parallelNodes, V, i)
6:      end for
7:      if isEmpty(TANodes) then
8:          return V
9:      else
10:         vS ← pickSourceNode(TANodes, ownPath, parallelNodes, V)
11:         vR ← minimumPathReduction(vS, ownPath, parallelNodes, V)
12:         add edge to V
13:         R_DC_DAG(V,m)
14:     end if
```

Figure 4.1d shows the refined DAG of the task presented in figure 4.1a. The *R_DC_DAG* algorithm added dispatch constraints between node $v4 \rightarrow v8$ and $v5 \rightarrow v8$. The result is the same number (two) of dispatch constraints added as for the *DC_DAG* algorithm, but they are added between different nodes. The difference is therefore that the longest path of the DAG is smaller for figure 4.1d compared to figure 4.1c (50 compared to 70), which is an improvement.

Algorithm 9 Computing the number of parallel paths

```
1: function computeNumPaths(parallelNodes_i, V, i)
2:     sinks = empty
3:     roots = empty
4:     for x ∈ parallelNode do
5:         sinks = sinks ∪ isempty(V(x).succ)
6:         roots = roots ∪ isempty(V(x).pred)
7:     end for
8:     pathNumber ∪ allpaths(roots, sinks)
9:     return pathNumber
```

The *computeNumPaths* algorithm takes the index i of the node for which the number of parallel paths should be computed as input. It also takes the DAG V and the set $parallelNodes_i$ for the selected node as input. The set $parallelNodes$ together with the structure of the DAG V provided is used to calculate the number of parallel paths. This is done by checking the nodes in the $parallelNode\_i$ set and putting the nodes with no successors (in the subgraph) into one set $sinks$, line 5. On line 6 the nodes in the $parallelNodes\_i$ set without predecessors in the subgraph are put into the $roots$ set. In Figure 4.1b the roots set would contain node $v3$ and the $sinks$ set would contain nodes $v8$ and $v9$. This is because it is between these "sink" and "root" nodes the number of paths should be calculated. The number of paths between each sink and root node is calculated and added to the integer $pathNumber$ (at line 8) and is then returned at line 9. To find all possible paths between two nodes the $ALLPATHS$ function from the "Introduction to Algorithms" textbook [13] was used.

4.3 Selecting the source node for the dispatch constraints

As previously mentioned, selecting the source node for the added dispatch constraints can be done in an infinite number of ways. The methods tested in this thesis are selected to keep the longest path of the DAG as short as possible while also utilizing as many of the available processors as possible. In this section, the different heuristics for selecting the source nodes for the dispatch constraints are presented and explained.

4.3.1 Biggest workload

The first method for selecting nodes is based on the workload of the nodes in the $TANodes$ set (see line 13 in algorithm 3). The node with the biggest workload is selected as the source. The motivation for this is that since all subtasks in the task have different-sized workloads it is preferable to schedule the nodes with the biggest workload as early as possible, because if they are not then the entire system could end up waiting for just one node to finish executing. This is to avoid situations like the one described in figure 2.2, where a node with a big workload is scheduled later (despite the total workload of the task being smaller). This could make the longest path in the DAG longer (unbalanced scheduling), while also leaving other processors in the system idle. The potential problem with this method is that it does not consider the structure of the DAG. For example, if the node with the biggest workload is placed at a deep level of the DAG using that node could create a very long longest path, because it could block a big subgraph of the task. The overall approach is presented in Algorithm 10.

Algorithm 10 Selecting the source node with the biggest workload

```
1: function pickSourceNode(TANodes, ownPath, parallelNodes, V)
2:     workload = empty
3:     for i = nodesinTANodes do
4:         workload = workload ∪ V(i).workload
5:     end for
6:     vS ← max(workload)
7:     return vS
```

Algorithm 10 works by, for each of the nodes in the $TANodes$ set (the nodes with more parallel nodes/paths than there are processors in the system), their workloads are put in the *workload* set (lines 3-5). The algorithm then selects the node with the biggest workload, from the *workload* set, as the source for the dispatch constraint. The selected node is returned, line 6.[1]

[1] the max function in line 6 in algorithm 10 returns the index of the node with the biggest workload.

4.3.2 Accumulated workload

Because the method of using the biggest workload to select the source node does not consider the internal structure of the DAG other options to select the source nodes were explored. This second approach examines the structure of the DAG, by taking the positions of the nodes in the paths into consideration. This solution works by adding the sum of the workloads of all the predecessors for each of the nodes (that have more parallel nodes/paths than processors in the system). The node with the smallest accumulative workload is then selected. The motivation for this method is that the sum of the workloads gives a better idea of where the nodes will be placed in the schedule, relative to each other. By selecting the node with the smallest total workload, situations, where a node deep down (close to the sink node) in the DAG are selected, can be avoided. The reason for wanting to avoid these situations is that they are more likely to result in longer paths in the DAG. Since the longest path in the DAG is the makespan, we want to avoid creating new long paths if it is not necessary. The approach is presented in Algorithm 11.

Algorithm 11 Selecting the source node with the smallest accumulative workload

```
1:  function pickSourceNode(TANodes, ownPath, parallelNodes, V)
2:      predecessors = empty
3:      accumulatedWorkloads = empty
4:      for i in TANodes do
5:          totalWorkload_i = empty
6:          predecessors = predecessors ∪ PredecessorPaths(V, i)
7:          for j in predecessors do
8:              totalWorkload_i = totalWorkload + V(j).workload
9:          end for
10:         accumulatedWorkload = accumulatedWorkloads ∪ totalWorkload_i
11:     end for
12:     vS ← min(accumulatedWorkloads)
13:     return vS
```

Selecting the source node based on the smallest accumulative workload works by first getting the predecessors for all the nodes in the $TANodes$ set, using the $PredecessorPaths$ function (on lines 4-6). The workload for all predecessors to the node is then accumulated on lines 7-9. The workload for each of the nodes in the $TANodes$ set is collected. The source node is then selected based on the smallest workload in that collection, line 12. The source node is returned at line 13.

4.4 Selecting the receiver node for the dispatch constraints

The methods for selecting the receiver nodes for the added dispatch constraints are with the same goal and motivation as for the source nodes. Keeping the longest path as short as possible and utilizing as many of the available processors as possible.

In this section, the different methods for selecting the receivers of the dispatch constraints are presented and described.

4.4.1 Smallest workload

The smallest workload method used for selecting the receiver node from the set of parallel nodes for the dispatch constraint is presented in Algorithm 3. The incentive of this approach is to avoid creating new longest paths in the DAG when adding the dispatch constraints. By selecting the receiver node based on the smallest workload making a relatively longer new longest path can be avoided. The function does not consider the workload of the other nodes in the paths created, meaning it is not guaranteed to not create new longest paths.

Algorithm 12 Selecting the receiver node with the smallest workload

```
1: function pickReceiverNode(vS, ownPath, parallelNodes, V)
2:     workload = empty
3:     for i in parallelNodes do
4:         workload = workload ∪ v(i).workload
5:     end for
6:     vR ← min(workload)
7:     return vR
```

The function works by iterating through the set of *parallelNodes* for the selected source node vS, line 3. For each of the nodes in the set the workload is put in the *workload* set (line 4). Once all the workloads have been collected in the *workload* set the node with the smallest workload is selected as the receiver node (vR), line 6. The chosen node is returned from the function at line 7.

4.4.2 The minimum path reduction

Recall that the aim of the refined Algorithm 8 is to consider the internal structure of the DAG when selecting the nodes to add dispatch constraint between the method described in this section is used. This function, described in detail in Algorithm 13, returns a receiver node selected based on the reduction of the number of parallel nodes/paths when a dispatch constraint is added from the source node vS.

Algorithm 13 Selecting the receiver node with the minimum path reduction

```
1: function MINIMUMPATHREDUCTION(vS, ownPath, parallelNodes, V)
2:     OG_parallelPaths ← COMPUTENUMPATHS(parallelNodes, V, vS)
3:     for x in parallelNodes do
4:         V_temp ←dispatch constraint from vS to x
5:         ownPath_temp ← SAMEPATH(V_temp, vS)
6:         parallelNodes_temp ← NOTSAMEPATH(V_temp, vS)
7:         TANodes_temp ∪ computeNumPaths(parallelNodes_temp, V_temp, vS)
8:     end for
9:     vR ←max(TANodes_temp) that is smaller than OG_parallelPaths
10:    return vR
```

Algorithm 13 takes the selected source node (for the dispatch constraint) vS, $ownPaths$, $parallelNodes$ and the DAG V as input. It returns the receiver node vS, which is selected based on how much a dispatch constraint minimizes the parallel nodes of vS. The function works by first calculating the number of parallel paths for the source node vS before adding any dispatch constraint, on line 2. It then iterates through each of the parallel nodes (line 3) and creates a new (temporary) DAG for each of them where a dispatch constraint is added between the source node vS and the parallel node. The number of parallel paths is then recalculated (lines 5-7) and saved in the $TANodes\_temp$ set for each such dispatch constraint. After this step has been done for all the parallel nodes the node with the smallest reduction in the number of paths, that is smaller than the original number of paths, is selected as the receiver node (line 9).

The motive for picking the receiver node this way is that the dispatch constraint is added to reduce the number of parallel paths/nodes which means it needs to be smaller than the original $OG\_parallelPaths$. At the same time (as mentioned earlier) the optimal case would be to utilize all processors at all time instants, meaning keeping the number of parallel nodes execution at each time instant as close to m as possible. Because of this, the number of parallel paths should not be reduced too much, which would happen if the maximum number of reductions were selected instead of the minimum. The receiver node is returned at line 10.

5

Experimental Setup

This section presents the setup used to evaluate the algorithms presented. The algorithms are applied to a DAG and the evaluation is based on whether or not the modified DAGs makespan meet the deadline. The purpose of the experimental setup is to create a simulation used to test each of the algorithms presented in this thesis and to compare it with the state-of-the-art algorithm. The simulator evaluates each heuristic for a certain number of iterations. For each iteration, the algorithms are executed with randomly generated DAGs. The algorithms will then return pass or fail depending on if the makespan meets the deadlines of the tasks. The algorithms will execute on the same randomly generated task sets to ensure a fair comparison.

5.1 Simulation method

The simulation tool provides several parameters that can be manually set to influence the system that is generated and evaluated. These simulation parameters are presented in the list below:

- m: number of processors the systems have.
- $Umax$: maximum value of utilization for the system, is the same as the number of processors simulated.
- $Umin$: minimum value of utilization for the system, the starting value.
- $stepU$: step in the variation of utilization.
- $Ucurr$: the current utilization value of the systems.

Before the algorithms can be simulated the task parameters need to be generated using the simulation parameters. The platform is generated based on the number of processors, which is set with the m parameter. The system utilization will range from 0 to m. At each utilization level, $Ucurr$, we generate 100 tasks. The parameter $Ucurr$ is then varied between $Umin$ and $Umax$ with the step size of $stepU$ ($Ucurr = Ucurr + stepU$).

5.1.1 DAG generation

The implementation for DAG generation is based on the work by Melani et al., see the paper [11] for further details. The DAG generation works by recursively

expanding non-terminal vertices to either terminal vertices or parallel subgraphs until a maximum recursion depth is reached (defined by the parameter *rec_depth*). The subgraphs are created with the recursion depth $rec\_depth - 1$ as a parameter. The task generation function takes the source and sink nodes as parameters. When a subgraph is created the node itself is the source node. The probability that a created vertex is a terminal node is determined by the p_{term} parameter and the probability that a vertex is a parallel subgraph is determined by the p_{par} parameter. The two parameters require that the condition $p_{par} + p_{term} = 1$ is fulfilled. The maximum number of parallel branches a subgraph can have is determined with the $maxParBranches$ parameter. For each subgraph created the number of parallel branches is randomly selected in the interval $[2, maxParBranches]$. When the nodes are created they get assigned a workload (WCET) randomly in the interval $[C_{min}, C_{max}]$.

Once all the nodes have been created and the maximum recursion depth is reached the resulting graph is a series-parallel graph. To make a DAG that better represents tasks with precedence constraints, there are additional edges added to the graph randomly. This is done based on the parameter *addprob*. The parameter determines the probability that an edge will be added between two vertices, providing the DAG requirements are still fulfilled (e.g. not creating any cycles).

The following list contains the previously mentioned task parameters, that can be adjusted to change the structure (and therefore the characterization) of the DAGs:

- *maxParBranches*: the maximum number of parallel branches for one node in the randomly generated DAG.
- *p_par*: probability that a generated node is a parallel subgraph.
- *p_term*: probability that a generated node is a terminal vertex.
- *Cmin*: minimum worst-case execution time for the nodes.
- *Cmax*: maximum worst-case execution time for the nodes.
- *rec_depth*: the recursion depth for the DAG.
- *addProb*: probability that a vertex will be added between two nodes.

6

Results

This chapter presents the results of the performed simulations. The results are compared to the state-of-the-art [11]. The number of task set that can be scheduled is used to measure the performance, given the multiprocessor platform. The results are presented in graphs, where the x-axis presents the utilization values *stepU* and the y-axis presents the number of schedulable task sets simulated (per each utilization step). For each utilization level, 100 task sets have been generated and applied to each of the algorithms. The number of schedulable task sets is counted and plotted in the graphs. To compare the different algorithms the number of schedulable task sets, also called the acceptance ratio, is used.

The performance is compared for the following four algorithms:

- Melani: the state-of-the-art by Melani et al. [11].
- DC_DAG, where the source node is picked based on the biggest workload (explained in section 4.3.1) and the receiver node is picked based on the smallest workload (explained in section 4.4.1).
- R_DC_DAG_accw, where the source node is picked based on the accumulated workload (explained in section 4.3.2) and the receiver node is picked based on the minimum path reduction (explained in section 4.4.2).
- R_DC_DAG_bigw, where the source node is picked based on the biggest workload (explained in section 4.3.1) and the receiver node is picked based on the minimum path reduction (explained in section 4.4.2).

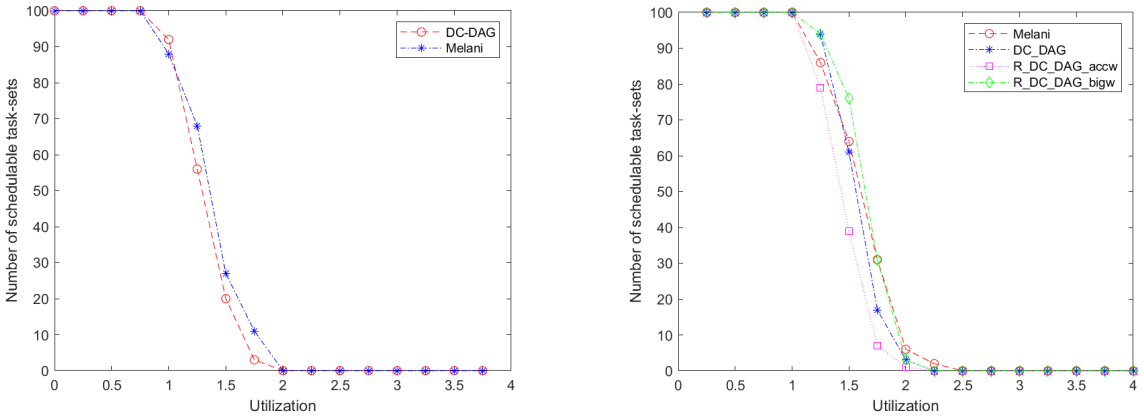
6.1 Results for different task structures

This section presents the results of the algorithms when varying both the simulation parameters and the task parameters for the randomly generated DAGs. Note that in this section each time a parameter is varied the other ones are kept fixed. The default values used are the following: $m = 4$, $maxParBranches = 4$, $p\_par = 0.6$, $p\_term = 0.4$, $Cmin = 1$, $Cmax = 100$, $rec\_depth = 2$ and $addProb = 0.4$.

Figure 6.1, with the corresponding tables 6.1 and 6.2, display the results for the algorithms when the makespan is calculated as the longest path of the dispatch-constrained DAGs. This is compared to the acceptance ratio of Melani's algorithm with its makespan calculation. These makespan calculations are timing anomaly

free and therefore guaranteed to be correct. In figure 6.1a, with only the *DC_DAG* and the Melani algorithms the observation can be made that the Melani algorithm performs better at most points. Although, at utilization level 1.25 the *DC_DAG* algorithm performs better. Meaning there are cases where the *DC_DAG* performs better than the state-of-the-art.

In figure 6.1b the other heuristics for adding dispatch constraints are presented. In figure 6.1b the *R_DC_DAG_bigw* heuristics has a higher acceptance rate than the other heuristics for most of the utilization values. This indicates that the method of selecting source and destination nodes creates DAG structures with a makespan less pessimistic than the state-of-the-art. The figure also shows that it is important to select the source and destination nodes such that the makespan is tighter. The *R_DC_DAG_bigw* algorithm performs the best, meaning creating structures where the nodes with big workloads are dispatched early results in a shorter makespan, thus utilizing the system resources more efficient.



(a) Makespan is the longest path.

(b) Makespan is the longest path.

Figure 6.1: Results without using simulation-based makespan computation, the makespan for the dispatch constraint algorithms is calculated based on the longest path. This figure is generated using values from table 6.1 and 6.2.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	88	68	27	11	0	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	92	56	20	3	0	0	0	0	0	0	0	0

Table 6.1: The number of schedulable task sets at each utilization level corresponding to the 6.1a figure. Parameter $m = 4$.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	86	64	31	6	2	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	94	61	17	3	0	0	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	76	39	7	1	0	0	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	94	76	31	3	0	0	0	0	0	0	0	0

Table 6.2: The number of schedulable task sets at each utilization level corresponding to the 6.1b figure. Parameter $m = 4$.

Presented next are the results where the makespan is computed using simulation. The following results assume the dispatch scheduling algorithm is timing anomaly free. As mentioned before, this has not been formally proven. A formal way of proving this would be adding new precedence constraints to the DAG based on how the nodes were scheduled in simulation so that there is a total ordering of the nodes on m processors.

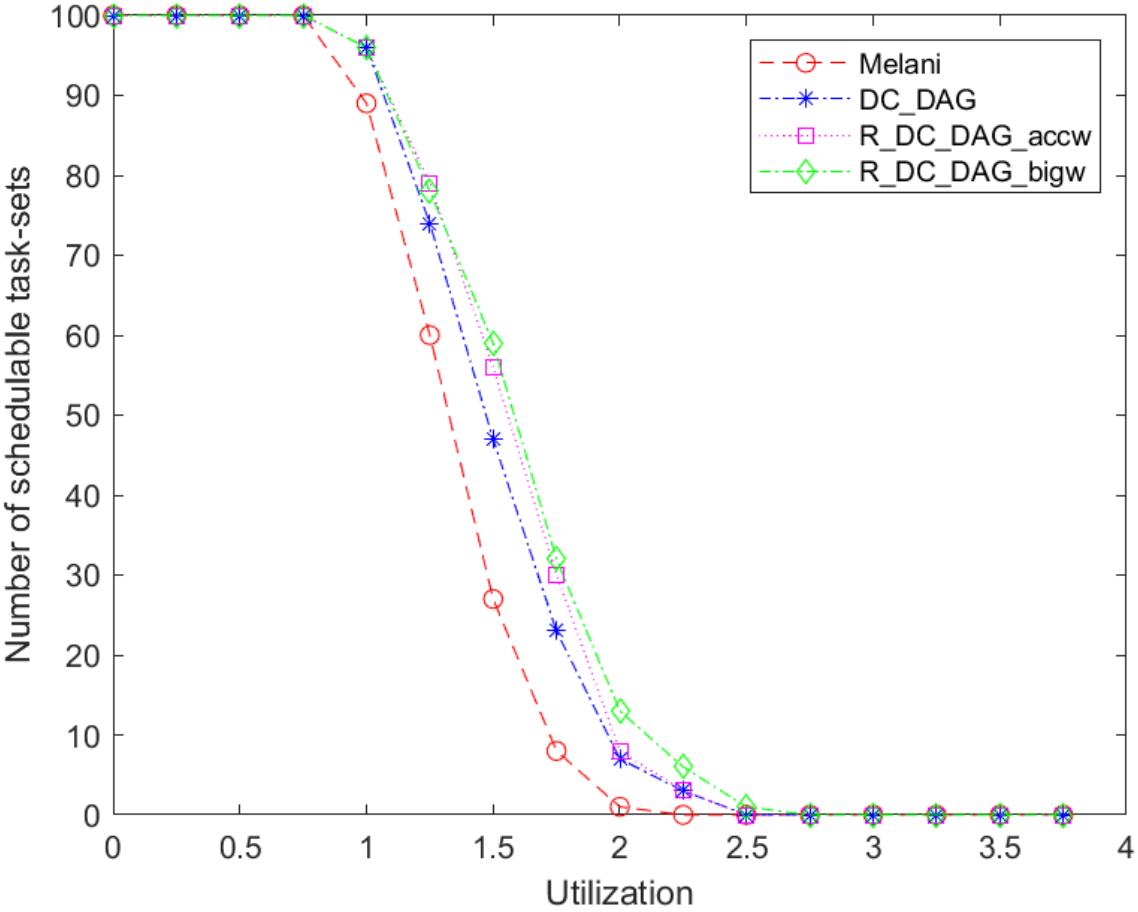


Figure 6.2: Comparison of the schedulability for the different algorithms based on makespan calculated with simulation scheduling, for the default settings of the system. Parameter $m = 4$.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	89	60	27	8	1	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	96	74	47	23	7	3	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	96	79	56	30	8	3	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	96	78	59	32	13	6	1	0	0	0	0	0

Table 6.3: The number of schedulable task sets at each utilization level for the different algorithms corresponding to figure 6.2.

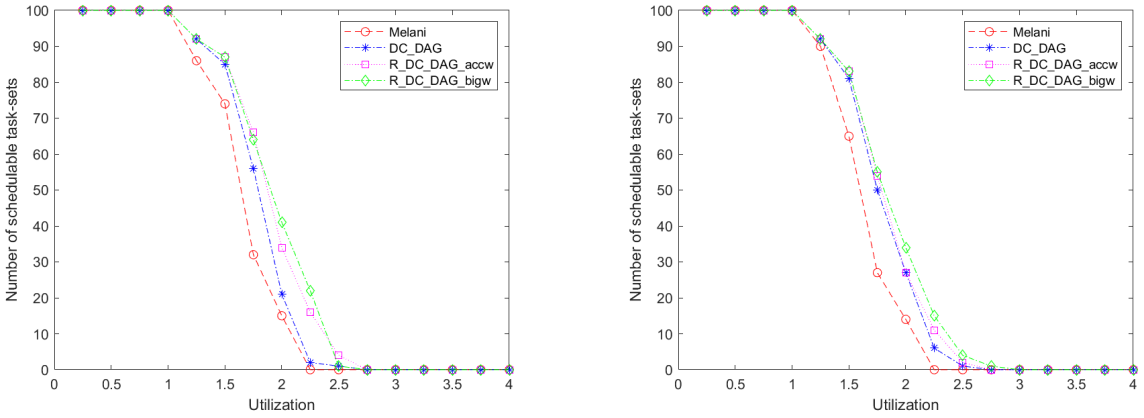
Figure 6.2 shows the results for the default setting of the simulation parameters, with the parameter values as per the description above. The general trend in the

figure shows that as the utilization increases the acceptance ratio of the task sets decreases, which is an expected trend because a higher utilization means a higher demand for the processing power and it is therefore natural for the schedulability to be dropping. However, comparing the state-of-the-art ('Melani' in the graph) with the heuristics-based algorithms, we can see that the dispatch constraint heuristics perform quite well.

Table 6.3 presents the number of task set, at each utilization level, that are marked as schedulable (out of 100) that figure 6.2 is based on. From this table, we can observe the difference in performance between the different algorithms. For example, at utilization level 1.75 the Melani algorithm has an acceptance rate of 27% while the best performing dispatch algorithm (*R_DC_DAG_bigw*) has an acceptance rate of 59%. This is an improvement of 32%. A reason for this could be the difference in calculating the makespan (based on simulation). Since all the *DC_DAG* heuristics use the simulation method to calculate the makespan, the simulation-based makespan could be a reason for such improvement.

The added dispatch constraints introduce predictability, which means the makespan can be calculated in a better way. The combination of adding dispatch constraints and using simulation-based makespan computation that utilized the processors efficiently to calculate the makespan seems to be working well. By adding dispatch constraints the DAGs can be restricted quite a lot, even making the level of parallelism less than the number of processors. However, the simulation method relaxes this because the dispatch constraints are not as strict as the traditional precedence constraints. The simulation balances out the restrictions made because of the different characteristics the dispatch constraints imply.

Figure 6.2 also shows that both versions of the *R_DC_DAG* algorithms (*R_DC_DAG_accw* and *R_DC_DAG_bigw*) perform better than the *DC_DAG* algorithm. The difference between the different versions of the algorithm is that the refined version considers the internal structure of the DAG, which is the reason for the difference in performance. Furthermore, looking at the two different versions of the refined algorithm (*R_DC_DAG*), a difference in performance can be observed. The *R_DC_DAG_bigw* algorithm performs better than the *R_DC_DAG_accw* algorithm, at almost all utilization levels in the figure. Since the only difference between these two algorithms is the method for selecting the source node to add the dispatch constraints, making it clear that the method for selecting the source node is critical for the performance. This could be because selecting based on the biggest workload means the "bigger" nodes are more likely to be scheduled earlier and therefore releasing a lot of parallelism and not becoming a sequential bottleneck, enabling the makespan to be calculated more efficiently.



(a) workload interval: [100-500]

(b) workload interval: [1-1000]

Figure 6.3: Acceptance rate for DAGs created with different workload intervals for the nodes.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	86	74	32	15	0	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	92	85	56	21	2	1	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	92	87	66	34	16	3	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	92	87	64	41	22	1	0	0	0	0	0	0

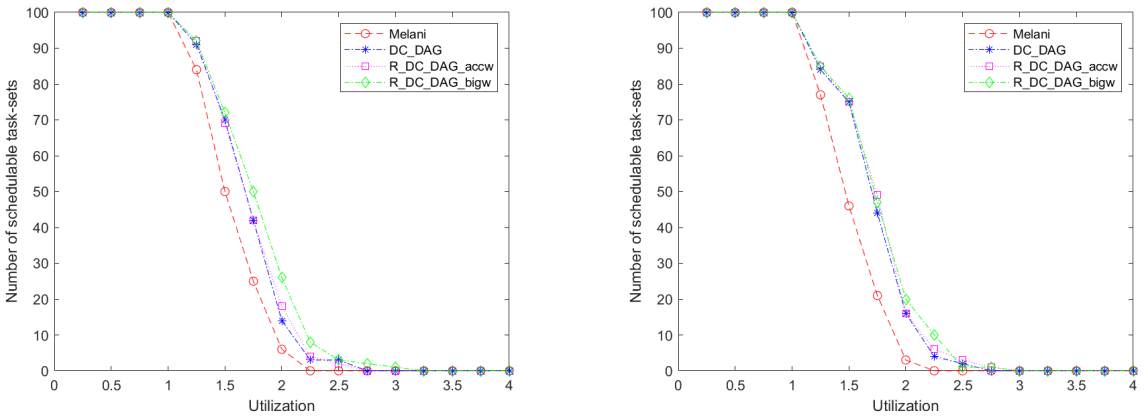
Table 6.4: Acceptance rate when the workload parameters are: $Cmin = 100$ and $Cmax = 500$.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	90	65	27	14	0	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	92	81	50	27	6	1	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	92	83	54	27	11	2	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	92	83	55	34	15	4	1	0	0	0	0	0

Table 6.5: Acceptance rate when the workload parameters are: $Cmin = 1$ and $Cmax = 1000$.

Figure 6.3 shows the impact of changing the interval of the workload. As mentioned earlier the workload for each node in the DAGs is randomly selected from this interval. The two graphs in the figure represent task sets where each node can get assigned a bigger workload, than the nodes in figure 6.2. Each node gets assigned a random value from the interval. The interval in figure 6.3a is $[100 - 500]$, and the interval in figure 6.3b is $[1 - 1000]$. Table 6.4 represents the acceptance rate for figure 6.3a and table 6.5 represent the acceptance rate for figure 6.3b. Both figures display similar results from the algorithms, relative to each other. From the presented data the observation can be made that the Melani algorithm produces similar results for both workload intervals. This shows that the makespan method is consistent for different workloads for the nodes. The dispatch algorithms show that both the *DC_DAG* and the *R_DC_DAG_bigw* algorithms produce better

results in figure 6.3b than figure 6.3a, while the *R_DC_DAG_accw* is the opposite. This can be because both *DC_DAG* and *R_DC_DAG_bigw* use the same method for picking the source node for the dispatch constraint, the available node with the biggest workload. When the workload interval is bigger, the method could have a more significant positive impact on the result.



(a) $p\_par = 0.5$, $p\_term = 0.5$ (b) $p\_par = 0.4$, $p\_term = 0.6$

Figure 6.4: Different ratios for generating parallel and terminal nodes.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	84	50	25	6	0	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	91	70	42	14	3	3	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	92	69	42	18	4	2	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	92	72	50	26	8	3	2	1	0	0	0	0

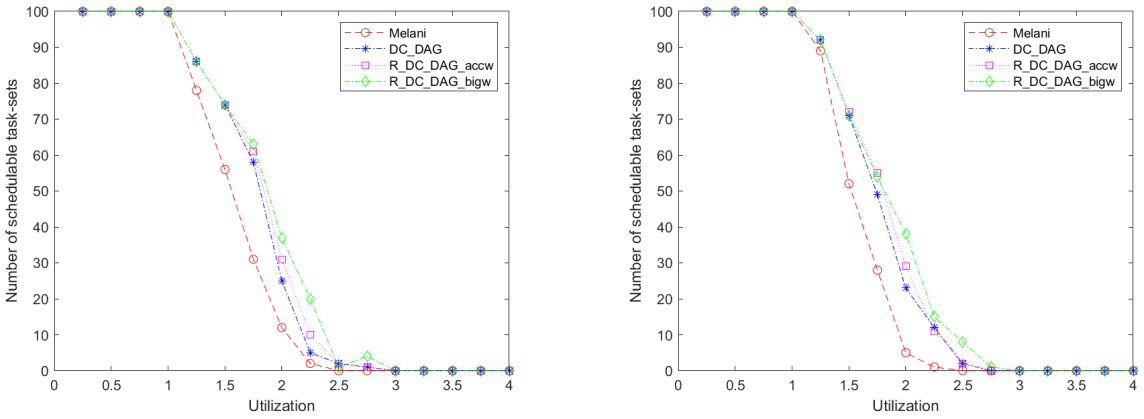
Table 6.6: Acceptance rate when the $p\_par = 0.5$ and $p\_term = 0.5$.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	77	46	21	3	0	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	84	75	44	4	2	0	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	85	75	49	16	6	3	1	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	85	76	47	20	10	1	1	0	0	0	0	0

Table 6.7: Acceptance rate when $p\_par = 0.4$, $p\_term = 0.6$.

Figure 6.4 shows the results for different ratios between the probability of a node being created as a parallel subgraph and a node being created as a terminal node when creating the DAGs. The parameters *p_par* and *p_term* are the ones changed, in figure 6.4a $p\_par = 0.5$ and $p\_term = 0.5$ and in figure 6.4b $p\_par = 0.4$ and $p\_term = 0.6$. Just as the previously presented results, also these figures show that the dispatch constraint algorithms achieve better results than the state-of-the-art. These results, just as figure 6.3 (workload interval), show that the parameter change does not affect the *Melani* algorithm much. The *Melani* algorithm produces very similar results for both figure 6.4a and for figure 6.4b. The dispatch constraint

heuristics, however, display similar results to each other. A reason for this could be that as the *p_term* parameter increases and the *p_par* parameter decreases, the number of parallel paths in the DAGs decreases. That, in its turn, means fewer dispatch constraints need to be introduced, which means the simulations will produce the same/very similar makespan calculations for the algorithms.



(a) $addProb = 0.2$ (b) $addProb = 0.3$

Figure 6.5: Different probability for adding random vertex between nodes.

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	78	56	31	12	2	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	86	74	58	25	5	2	1	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	86	74	61	31	10	2	1	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	86	74	63	20	10	1	4	0	0	0	0	0

Table 6.8: Acceptance rate when the $addProb = 0.2$

Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	89	52	28	5	1	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	92	71	49	23	12	2	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	92	72	55	29	11	2	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	92	71	54	38	15	8	1	0	0	0	0	0

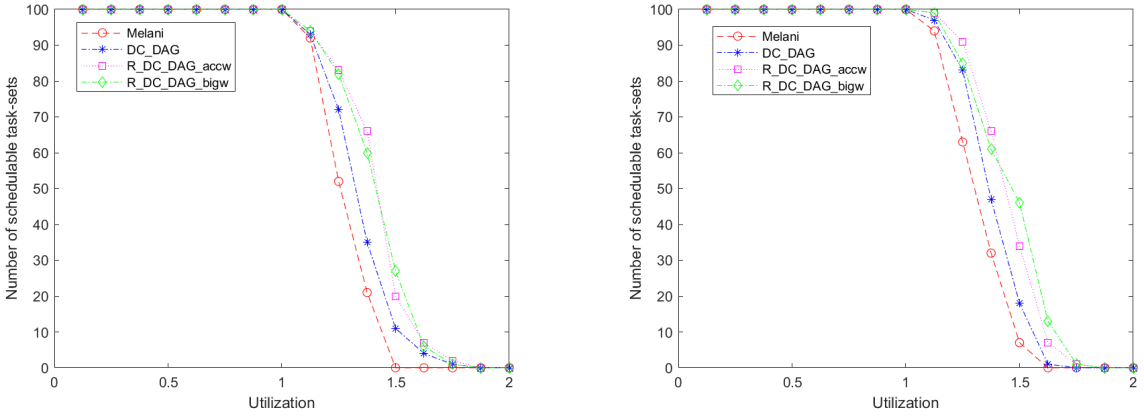
Table 6.9: Acceptance rate when the $addProb = 0.3$

Figure 6.5 shows the results of adjusting the *addProb* parameter. This parameter controls the probability that a precedence constraint will be added between two nodes randomly when creating the DAGs. This affects the structure of the original DAG. As the probability increases more precedence constraints will be added, which creates more paths in the DAG. Table 6.8 and 6.9 display the acceptance rate figures 6.5a and 6.5b are based on. The figures show similar relation between the algorithms for the results. The dispatch algorithms perform better than the state-of-the-art. There is not a noticeable difference in the performance for any of the algorithms as the *addProb* parameter increases.

6.2 Results for different system models

In this section, the presented results represent systems with a different number of processors. This is done by changing the parameter for the number of processors m, while also adjusting the task parameter for the maximum number of parallel branches ($maxParBranches$) that is used when randomly generating the DAGs. As the number of parallel branches increases so will the probability that the DAGs have more parallel paths. Note that these results are with the simulation-based makespan computation.

Figure 6.6 show the results for systems with two processors $m = 2$. Figure 6.6a and table 6.10 display the acceptance rate of the algorithms when the number of max parallel branches is three $maxParBranches = 3$. Figure 6.6b and 6.11 display the acceptance ration when $maxParBranches = 4$. The algorithms show similar trends, relative to each other, as the previously presented results when $maxParBranches$ varies.



(a) $maxParBranches = 3$ and $m = 2$ (b) $maxParBranches = 4$ and $m = 2$

Figure 6.6: Results for when parameter $m = 2$ and parameter $maxParallelBranches$ varies.

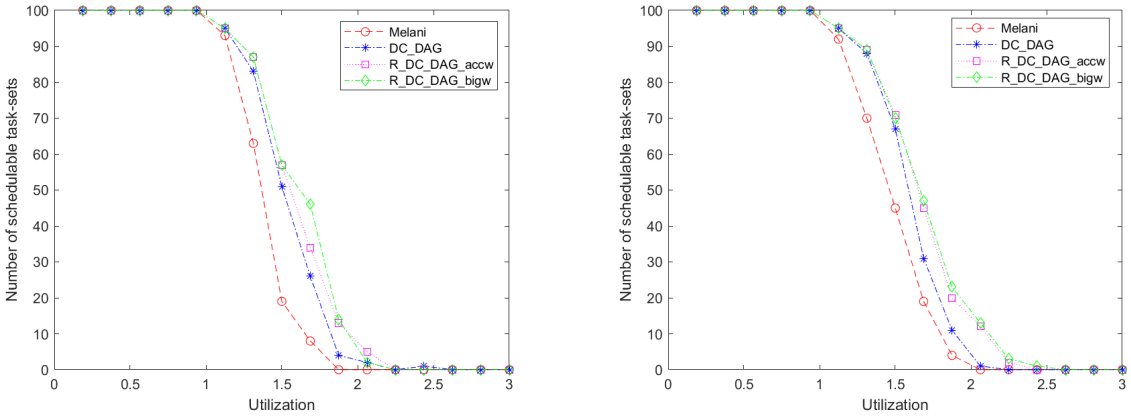
Algorithms \ Utilization	0,125	0,25	0,375	0,50	0,625	0,75	0,875	1,00	1,125	1,25	1,375	1,50	1,625	1,75	1,875	2,00
Melani	100	100	100	100	100	100	100	100	92	52	21	0	0	0	0	0
DC_DAG	100	100	100	100	100	100	100	100	93	72	35	11	4	1	0	0
R_DC_DAG_accw	100	100	100	100	100	100	100	94	83	66	20	7	2	0	0	0
R_DC_DAG_bigw	100	100	100	100	100	100	100	100	94	82	60	27	6	1	0	0

Table 6.10: $maxParBranches = 3$ and $m = 2$

Algorithms \ Utilization	0,125	0,25	0,375	0,50	0,625	0,75	0,875	1,00	1,125	1,25	1,375	1,50	1,625	1,75	1,875	2,00
Melani	100	100	100	100	100	100	100	100	94	63	32	7	0	0	0	0
DC_DAG	100	100	100	100	100	100	100	100	97	83	47	18	1	0	0	0
R_DC_DAG_accw	100	100	100	100	100	100	100	100	99	91	66	34	7	1	0	0
R_DC_DAG_bigw	100	100	100	100	100	100	100	100	99	85	61	46	13	1	0	0

Table 6.11: $maxParBranches = 4$ and $m = 2$

The same goes for the other results in figures 6.7 (with tables 6.12 and 6.13) and 6.8 (with tables 6.14 and 6.15), that represent systems with $m = 3$ respectively $m = 4$ processors. There is no unexpected variation in any of the systems, which indicates that the algorithms are not sensitive to more or less parallelism. In figure 6.8 the acceptance rate for the algorithms are closer to each other, for each utilization step. The probable cause of this is that as the number of processors increases the need for parallelism decrease, meaning timing anomalies are less likely to occur. This means, the algorithms do not adjust the DAGs and therefore produce the same makespan as each other more often.



(a) $maxParBranches = 3$ and $m = 3$ (b) $maxParBranches = 4$ and $m = 3$

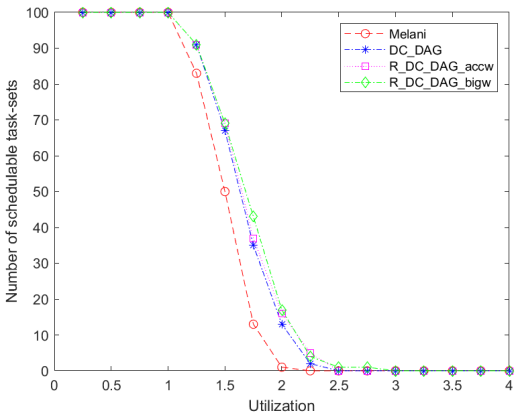
Figure 6.7: Results for when parameter $m = 3$ and parameter $maxParallelBranches$ varies.

Algorithms \ Utilization	0,1875	0,375	0,5625	0,75	0,9375	1,125	1,3125	1,50	1,6875	1,875	2,0625	2,25	2,4375	2,625	2,8125	3,00
Melani	100	100	100	100	100	93	63	19	8	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	100	95	83	51	26	4	2	0	1	0	0	0
R_DC_DAG_accw	100	100	100	100	100	95	87	57	34	13	5	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	100	95	87	57	46	14	2	0	0	0	0	0

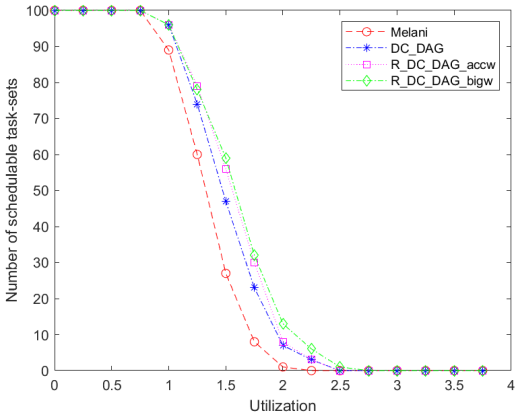
Table 6.12: $maxParBranches = 3$ and $m = 3$

Algorithms \ Utilization	0,1875	0,375	0,5625	0,75	0,9375	1,125	1,3125	1,50	1,6875	1,875	2,0625	2,25	2,4375	2,625	2,8125	3,00
Melani	100	100	100	100	100	92	70	45	19	4	0	0	0	0	0	0
DC_DAG	100	100	100	100	100	95	88	67	31	11	1	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	100	95	89	71	45	20	12	2	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	100	95	89	70	47	23	13	3	1	0	0	0

Table 6.13: $maxParBranches = 4$ and $m = 3$



(a) $maxParBranches = 3$ and $m = 4$



(b) $maxParBranches = 4$ and $m = 4$

Figure 6.8: Results for when parameter $m = 4$ and parameter $maxParallelBranches$ varies.

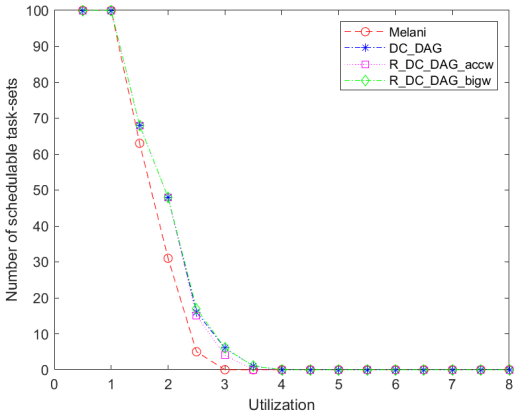
Algorithms \ Utilization	0,25	0,50	0,75	1,00	1,25	1,50	1,75	2,00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	83	50	13	1	0	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	91	67	35	13	2	0	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	91	69	37	16	5	0	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	91	69	43	17	4	1	1	0	0	0	0	0

Table 6.14: $maxParBranches = 3$ and $m = 4$

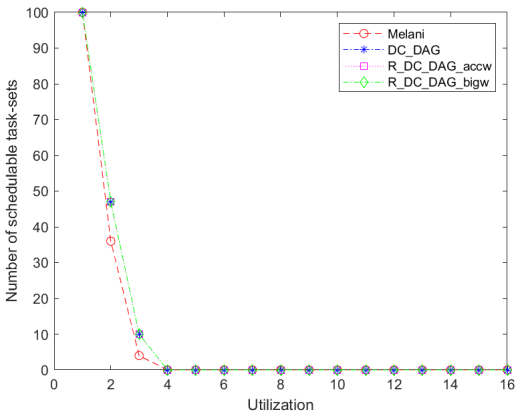
Algorithms \ Utilization	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Melani	100	100	100	100	89	60	27	8	1	0	0	0	0	0	0	0
DC_DAG	100	100	100	100	96	74	47	23	7	3	0	0	0	0	0	0
R_DC_DAG_accw	100	100	100	100	96	79	56	30	8	3	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	100	100	96	78	59	32	13	6	1	0	0	0	0	0

Table 6.15: $maxParBranches = 4$ and $m = 4$

Figure 6.9 shows the results for systems with $m = 8$ and $m = 16$ (with tables 6.16 and 6.17). Like the previously presented results, the systems with a bigger number of processors are predictable, the results do not vary much.



(a) $maxParBranches = 4$ and $m = 8$



(b) $maxParBranches = 4$ and $m = 16$

Figure 6.9: Results for when parameter $m = 4$ varies and parameter $maxParallelBranches = 4$.

Algorithms \ Utilization	0,5	1,0	1,5	2,0	2,5	3,0	3,5	4,0	4,5	5,0	5,5	6,0	6,5	7,0	7,5	8,0
Melani	100	100	63	31	5	0	0	0	0	0	0	0	0	0	0	0
DC_DAG	100	100	68	48	16	6	1	0	0	0	0	0	0	0	0	0
R_DC_DAG_accw	100	100	68	48	15	4	0	0	0	0	0	0	0	0	0	0
R_DC_DAG_bigw	100	100	68	48	17	6	1	0	0	0	0	0	0	0	0	0

Table 6.16: $maxParBranches = 4$ and $m = 8$

Algorithms \ Utilization	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	10,0	11,0	12,0	13,0	14,0	15,0	16,0
Melani	100	36	4	0	0	0	0	0	0	0	0	0	0	0	0	0
DC_DAG	100	47	10	0	0	0	0	0	0	0	0	0	0	0	0	0
R_DC_DAG_accw	100	47	10	0	0	0	0	0	0	0	0	0	0	0	0	0
R_DC_DAG_bigw	100	47	10	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6.17: $maxParBranches = 4$ and $m = 16$

6.3 Sensitivity analysis

This section compares the acceptance rate of the previously presented results, for each of the algorithms. This is to see how sensitive the algorithms are to changes in the systems and the structure of the tasks, relative to itself. The tables in this section collect the algorithms' results separately from the tests where the system parameters are varied.

The first algorithm *DC_DAGs* results are presented in table 6.18. The table shows the same trend in the numbers, starting high and then decreasing, for the different utilization steps. The table shows that the parameters changing does not affect the algorithm in any specific way. The different values for each of the parameters show fluctuating numbers of schedulable task sets relative to each other. For example, the acceptance ratio of the *DC_DAG* at utilization level 1.5 is 74% and 71% respectively for $addProb = 0.2$ and $addProb = 0.3$. One parameter value is not consistently producing higher or lower acceptance rate values than the other, for any of the parameters. This indicates that the algorithm is not sensitive to system changes, which shows the scalability/robustness of the algorithm.

DC_DAG	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Default	100	100	100	100	96	74	32	7	3	0	0	0	0	0	0	0
Cmin=100, Cmax=500	100	100	100	100	92	85	56	21	2	1	0	0	0	0	0	0
Cmin=1, Cmax=1000	100	100	100	100	92	81	50	27	6	1	0	0	0	0	0	0
p_par=0.5, p_term=0.5	100	100	100	100	91	70	42	14	3	3	0	0	0	0	0	0
p_par=0.4, p_term=0.6	100	100	100	100	84	75	44	4	2	0	0	0	0	0	0	0
addProb=0.2	100	100	100	100	86	74	58	25	5	2	1	0	0	0	0	0
addProb=0.3	100	100	100	100	92	71	49	23	12	2	0	0	0	0	0	0

Table 6.18: Data for the *DC_DAG* algorithm.

The *R_DC_DAG_accw* algorithms results are displayed in table 6.19 and show similar robustness against changing parameters as the *DC_DAG* algorithm, with one exception. The table shows that changing the workload interval ($Cmin$ and $Cmax$ values) consistently affects the result. When $Cmin = 100$ and $Cmax = 500$ the acceptance ratio is higher for all utilization values relative to when $Cim = 1$ and $Cmax = 1000$. The reason for this could be that the nodes being assigned workloads from a bigger interval means the schedule is more likely to be unbalanced at the end, with multiple processors being idle while the systems "wait" for one node with a large workload to finish executing. This relation between the different workload intervals is reasonable and expected because of the non-preemptive scheduling.

R_DC_DAG_accw	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Default	100	100	100	100	96	79	56	30	8	3	0	0	0	0	0	0
Cmin=100, Cmax=500	100	100	100	100	92	87	66	34	16	3	0	0	0	0	0	0
Cmin=1, Cmax=1000	100	100	100	100	92	83	54	27	11	2	0	0	0	0	0	0
p_par=0.5, p_term=0.5	100	100	100	100	92	69	42	18	4	2	0	0	0	0	0	0
p_par=0.4, p_term=0.6	100	100	100	100	85	75	49	16	6	3	0	0	0	0	0	0
addProb=0.2	100	100	100	100	86	74	61	31	10	2	1	0	0	0	0	0
addProb=0.3	100	100	100	100	92	72	55	29	11	2	0	0	0	0	0	0

Table 6.19: Data for the *R_DC_DAG_accw* algorithm.

The results presented in table 6.20 correspond to the *R_DC_DAG_bigw* algorithm. Just like the *DC_DAG* algorithm, this algorithm too is not sensitive to the parameters changing in the system. The two algorithms displaying similar behavior could be because the selection method for the source node to add the dispatch constraints from is the same.

R_DC_DAG_bigw	0,25	0,50	0,75	1.00	1,25	1,50	1,75	2.00	2,25	2,50	2,75	3,00	3,25	3,50	3,75	4,00
Default	100	100	100	100	96	78	59	32	13	6	1	0	0	0	0	0
Cmin=100, Cmax=500	100	100	100	100	92	87	64	41	22	1	0	0	0	0	0	0
Cmin=1, Cmax=1000	100	100	100	100	92	83	55	34	15	4	1	0	0	0	0	0
p_par=0.5, p_term=0.5	100	100	100	100	92	72	50	26	8	3	2	0	0	0	0	0
p_par=0.4, p_term=0.6	100	100	100	100	85	76	47	20	10	1	1	0	0	0	0	0
addProb=0.2	100	100	100	100	86	74	63	20	10	1	4	0	0	0	0	0
addProb=0.3	100	100	100	100	92	71	54	38	15	8	1	0	0	0	0	0

Table 6.20: Data for the *R_DC_DAG_bigw* algorithm.

7

Conclusion, Discussion and Future work

This chapter contains conclusion, discussion of the results and limitations of the thesis including direction for future work.

7.1 Discussion, Limitations and Conclusion

In this thesis, the possibility to improve the makespan calculation of parallel tasks for multi-processor real-time systems has been explored. This has been done by introducing dispatch constraints among the subtasks of parallel tasks. The dispatch constraints have different properties compared to traditional precedence constraints. The constraints are added until there are no points in the task where the number of parallel nodes, in the DAG, is bigger than the number of available processors in the system. A heuristic method is used to introduce the constraints. The different heuristics are used to evaluate the method's effectiveness.

The work presented in this thesis shows that introducing additional constraints to tasks does impact the makespan calculations. Comparing the results of the presented algorithms also show that considering the internal structure matters for computing the makespan tightly. In most of the tests, both of the refined *R_DC_DAG* algorithms performed the best, leading to the conclusion that the method of selecting the destination node for the dispatch constraints based on the minimum path reduction in combination with the simulation-based makespan computation, creates the most balanced DAGs and the tightest overall makespan. Here the term "balanced" means, restricting the DAG structure while also loosening it with the simulation schedule.

One aspect of the presented algorithms, that could be considered a limitation, is the cases when multiple nodes can be selected. For example, when multiple nodes have the same biggest workload when the source node for the dispatch constraint can be selected in multiple ways. In those cases, the current implementation selects the node with the smallest index in the DAG. There is no evaluation or comparison of the possibilities. Since the results of the different algorithms conclude that the heuristics do impact the results, we can assume that also this is an aspect that may affect the results and should therefore be considered.

Although a prof sketch for the simulation-based makespan calculation is given in this thesis a formal proof is left as future work. The makespan produced by only considering the longest paths in the transformed DAGs does not perform better than the state-of-the-art in most cases. Although these results do confirm that the heuristic method does have a positive impact on the acceptance ratio in some cases. Each algorithm produces different results, indicating that development and further exploration of the heuristics could perform better.

Another limitation of the results is the fact that the randomly generated DAGs and the experimental setup do not necessarily reflect real programs and tasks. Meaning the algorithms have not been tested on real data. No conclusions on their performance on real data can therefore be made. However, the results from changing the parameters do confirm that the algorithms are not sensitive to different types of system configurations. They produce similar results for most changes to the systems. This allows for the assumption that the algorithms would also perform similarly on data representing real parallel programs.

7.2 Future work

As stated in section 3.2.1 the work in this thesis assumes the simulation algorithm for the DAGs with dispatch constraints is timing anomaly free. However, formal proof of this is needed, but because of the time limitations of the project, this could not be provided in this thesis. Therefore, proof that the simulation scheduling is timing anomaly free is left for future work.

In this thesis, the heuristic method was chosen because the optimal option (testing all possible combinations of nodes to add the dispatch constraints) is not computationally feasible. Testing all possible ways of adding dispatch constraints to a DAG takes exponential time, while the heuristic method takes polynomial time. Because of this, there are an endless number of heuristics that could be tested. Based on the work in this thesis evaluating more of those possibilities is a future work. An example is to add dispatch constraints between two groups of nodes rather than from individual nodes.

Future work could also include extending the implementation to consider heterogeneous processor systems. The implementation in this thesis only considers homogeneous processor systems. The challenge with heterogeneous systems is to calculate the makespan and scheduling when the processors operate at different speeds for different subtasks.

In this thesis we have provided a framework for computing the makespan of a single DAG on a multiprocessor platform. Many other research areas such as federated scheduling and scheduling of mixed-criticality parallel tasks use the makespan of a single task to do a schedulability analysis for multiple tasks. Therefore the results of this thesis can also be applied to other domains such as federated scheduling and mixed-criticality scheduling. Extending the work of this thesis to these domains is possible for future work.

Bibliography

[1] T. Abdelzaher, E. M. Atkins, and K. G. Shin, "Qos negotiation in real-time systems and its application to automated flight control," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1170–1183, 2000.

[2] A. Elewi, M. Shalan, M. Awadalla, and E. M. Saad, "Energy-efficient task allocation techniques for asymmetric multiprocessor embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2s, pp. 1–27, 2014.

[3] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 4, pp. 473–487, 1989.

[4] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of np-completeness*. W.H Freeman and Company, San Francisco, 1979.

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

[6] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *2014 26th Euromicro Conference on Real-Time Systems*, IEEE, 2014, pp. 85–96.

[7] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*, IEEE, 2010, pp. 259–268.

[8] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol. 49, pp. 404–435, 2013.

[9] 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.

[10] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho, "A multi-dag model for real-time parallel applications with conditional execution," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 1925–1932.

[11] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *2015 27th Euromicro Conference on Real-Time Systems*, IEEE, 2015, pp. 211–221.

[12] P. Voudouris, P. Stenström, and R. Pathan, "Timing-anomaly free dynamic scheduling of task-based parallel applications," in *2017 IEEE Real-Time and*

Embedded Technology and Applications Symposium (RTAS), IEEE, 2017, pp. 365–376.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.