FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automatisierte Analyse von Echtzeit Scheduling Algorithmen

für feste Echtzeitanforderungen mit nicht-präemptiven Bereichen und Reihenfolgenbeschränkungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Nico Schaumberger, BSc
Matrikelnummer 01025736

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Ulrich Schmid

Wien, 7 October, 2019

Nico Schaumberger

Ulrich Schmid

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Automatic competitive analysis of real-time scheduling algorithms

for firm-deadline tasks with non-preemptible sections and precedence constraints

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Nico Schaumberger, BSc
Registration Number 01025736

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Ulrich Schmid

Vienna, 7th October, 2019

Nico Schaumberger | Ulrich Schmid

Erklärung zur Verfassung der Arbeit

Nico Schaumberger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7 October, 2019

Nico Schaumberger

Acknowledgements

I am indebted to Andreas Pavlogiannis (Aarhus University) and to Krishnendu Chatterjee (IST Austria) their support in understanding some details of the previous implementation of the framework, and for their suggestions for and feedback on the work described in this thesis. I would also like to thank my friends, my family and my supervisor, Prof. Ulrich Schmid.

Kurzfassung

Diese Diplomarbeit behandelt die automatische Berechung der competitive ratio von online Schedulung Algorithmen mit firmen Echtzeitanforderungen und nicht-präemptiven Bereichen und Reihenfolgenbeschränkungen. Zu diesem Zweck wurde ein existierendes Framework [CPKS18] um diese Funktionalität erweitert. Dabei mussten beträchtliche theoretische und implementierungstechnische Herausforderungen bewältigt werden. Zum Einen impliziert das Vorliegen von Event- und/oder Zeit-basierten Reihenfolgenbeschränkungen, dass der Online- und der Offline-Algorithmus bei der Berechnung der competitive ratio nicht notwendigerweise dieselbe job sequence verarbeiten. Zum Anderen ist EDF unter Reihenfolgenbeschränkungen nicht mehr optimal, wodurch die ursprüngliche Implementierung des Offline-Algorithmus nicht mehr verwendbar war, sondern basierend auf EDF* neu entwickelt werden musste. Schlussendlich wurden Teile des Algorithmus parallelisiert und in CUDA implementiert. Mit der parallelen Rechenleistung einer GPU resultiert daraus eine Performance-Verbesserung um mehrere Größenordnungen, sodass nunmehr auch größere und komplexere task sets analysiert werden können.

Abstract

This thesis deals with automatically computing the competitive ratio of online real-time scheduling algorithms for firm deadline task with non-preemptible sections and precedence constraints. For this purpose, an existing framework for automatic competitive analysis [CPKS18] is extended with the needed functionality. Several challenges, both theoretical and implementation-wise, make this very difficult. First of all, event-based and time-based precedences require online and offline algorithms to work on different job sequences when computing the competitive ratio. Moreover, as EDF is no longer optimal in the presence of precedences, we could not use the original implementation of the offline algorithm but had to develop a new one based on EDF*. Finally, in order to be able to analyze more complex task sets, parts of the algorithm are implemented in CUDA. Leveraging the parallel processing power of a GPU, this increases the achieved performance of the framework by several orders of magnitude.

Contents

CHAPTER 1

Introduction

1.1 Real-time scheduling

The field of real-time scheduling deals with modeling and analysis of real-time scheduling algorithms. A real-time scheduling algorithm operates on a set of real-time task instances (also known as jobs), which are released by an adversary, and decides which task instances are executed. In this thesis, we will only consider uniprocessor scheduling. In what follows, we will briefly introduce the most important terminology; consult [SAr$^+$04] for a comprehensive overview.

A task set $\{\tau_0, \tau_1, ...\tau_n\}$ is a set of tasks, and the jobs $J_{i,j}$ with $j \geq 1$ that may be released are instances of task τ_i. A real-time task is characterized by various properties, depending on the framework used to model them. In [CPKS18], the relevant properties of a task τ_i are its computation time C_i, its relative deadline D_i and its utility V_i. The computation time C_i of a task is given as the number of "time-slots" this task needs to be executed for in order to be completed. A time-slot is the smallest unit of time the model considers. Therefore, all durations of time can be given as a number of time-slots, and all points in time can be given as the number of time-slots after the start of the execution. The release-time of a task instance is the first time-slot where this task instance can be executed. This release-time plus the relative deadline is the first time-slot where this task-instance can no longer be executed, also known as the (absolute) deadline of this task instance.

There are various ways to categorize the type of tasks studied in real-time scheduling: The deadline of a task can be hard, firm or soft. A hard deadline task must be finished by its deadline, or the whole system will suffer a catastrophic failure. If a firm deadline task is not finished by its deadline, then the result will be useless, but it is not a catastrophe. The result of a soft deadline task might still be useful for some time even after the deadline is passed, so it may continue execution even after the deadline.

The scheduling environment of a task can be preemptive or non-preemptive. In preemptive scheduling, a currently executed job can be suspended (preempted) in order to execute a different job, while in non-preemptive scheduling a job cannot be suspended until it is completed.

Multiple tasks might be independent or precedence-ordered. Independent task instances can be executed in any order, while task instances with a precedence-order must be executed in that order, so a task instance can only be scheduled when all task instances it depends on are already completed. When multiple tasks need to access some resource, but the resource can only be used by one task at a time, then these tasks are under a mutual exclusion constraint. This means that once any task instance has entered the section in its execution where the resource is needed, which is called the "critical section", then no other task instance may enter its critical section until this first task instance has finished its critical section and released the resource.

Overall, the real-time scheduling problem can be understood as a game between the adversary, who generates task instances over time, and the scheduling algorithm, who tries to schedule all released tasks in order to achieve certain goals. In hard real-time scheduling, all instances must complete by the deadline. In firm-deadline scheduling, a task instance that is not completed by its deadline does no harm, but also does not add any utility to the system. A task instance that is completed by its deadline does add its utility, and the analyzed scheduling algorithms are evaluated with respect to how good they are at maximizing the collected utility.

Since it is impossible to feasibly schedule all jobs generated by an unpredictable adversary over time, its behavior is typically severely constrained. In particular, the releases of a task can be periodic, sporadic or aperiodic. For a periodically released task, the release of a task instance always happens a fixed time after the previous task instance has been released. For sporadic tasks, new task instances cannot be released for a certain time after the last task instance has been released, but they do not have to be released when they could be released for the first time. Instances of aperiodic tasks can be released at any time, regardless of when the last instance was released.

Less standard examples of safety constraints would be a bounded load density or given number of slots or some job release order constraints. Other types of constraints are liveness constraints, which e.g. require the adversary to generate some task infinitely often. Finally, there may be (limit) average constraints, like a bounded average load of the generated jobs.

Some well known scheduling algorithms are EDF (earliest deadline first - always schedules the job with the earliest absolute deadline), FIFO (first in, first out - always schedules the job with the earliest release time), and SP (static priorities - always schedules the job of the highest priority task, and task priorities are constant). Figure 1.1 provides an illustration of an example EDF-schedule, used in [But11].

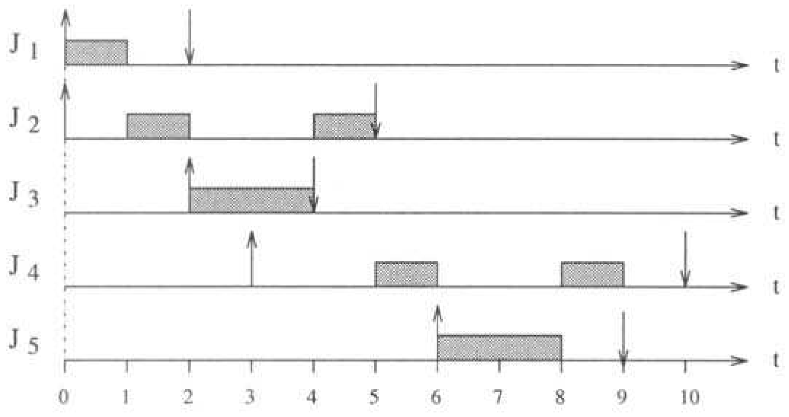

Figure 1.1: Example of an EDF-schedule [But11]. J_i means jobs of τ_i, $\uparrow$ means the release of a job, $\downarrow$ means the absolute deadline of a job.

1.2 Competitive analysis

In general, an algorithm makes better decisions if it has access to more information: Algorithms operating with incomplete information can lead to less than optimal results, especially if previous decisions cannot be undone as more information becomes available. The field of competitive analysis [BEY98] deals with assessing the performance of these algorithms operating on incomplete information (also called "online algorithms") by comparing them with the performance of an optimal algorithm that has complete information (also called an "offline algorithm" or "clairvoyant algorithm"). The performance of these algorithms is usually measured in terms of some generated utility (also called value). The competitive ratio of an online algorithm is the worst case of the online algorithm's collected utility divided by the offline algorithm's collected utility, evaluated over all possible inputs of the given problem. A large competitive ratio means that the online algorithm is able to cope well with the limited information it is given, while a small competitive ratio means the online algorithm performs badly due to its limited information.

This comparison between the online algorithm and the offline algorithm can be analyzed like a 2-player game, where player 1 is the online algorithm, and player 2 (also called the adversary) controls the offline algorithm and also chooses the input on which both algorithms operate. Each move of player 2 consists of choosing the input for a unit of computation as well as how the offline algorithm responds to this input. Then, each move of player 1 is the response of the online algorithm to the input chosen by player 2. The objective of player 2 is to minimize the competitive ratio, which means that player 2 chooses the inputs in such a way that the collected utility of the online algorithm divided by the collected utility of the offline algorithm becomes as small as possible. For this purpose, player 2 is assumed to be clairvoyant, that is, player 2 knows what player 1 will do in each possible situation. This framework allows the application of methods from the field of game theory to the analysis of the performance of online algorithms.

Using competitive analysis to judge the performance of real-time scheduling algorithms is a common technique since [BKM+92]. In the context of real-time scheduling, player 1 is the scheduling algorithm, which has incomplete information about the future task releases chosen by player 2. The automatic competitive analysis framework [CPKS18] computes the competitive ratio of an online algorithm A on a given task set $\mathcal{T}$ of firm-deadline, preemptive, independent, aperiodic tasks, with optional safety, liveness and limit-average constraints.

1.3 Problem description

In this work, the automatic competitive analysis framework [CPKS18] for real-time scheduling algorithms for firm-deadline tasks is expanded to include non-preemptible sections and precedence constraints.

1. In particular, our extension also includes tasks that might be completely or partially non-preemptible, or have complex precedence relations. A partially-preemptible task has at least non-preemptible section, which means that while the currently running job is in this non-preemptible section, it cannot be preempted. Note that this allows to implement mutual exclusion constraints, albeit in a somewhat simplistic way. A precedence relation means that an instance of a dependent task can only be released after instances of a certain set of other tasks have been completed. The framework presented in this thesis allows for both event-based dependencies, where an instance of the dependent task must be released immediately after the required task instances have been completed, as well as for time-based dependencies, where the dependent task could also be released later on.

2. The goal remains to determine the competitive ratio of an online algorithm on a given task set, now under these new conditions. Non-preemptible sections and precedence constraints are relevant in various situations, like query scheduling [ZWL13] and scheduling disk requests [PKB+08]. Existing models for applications like operator-scheduling, stop-and-go network switching, interrupt-handling or network-on-chip scheduling can also potentially be analyzed with this more expressive view on the scheduling problem: [BBMD03] [TBW94] [BIS10].

3. Finally, in order to speed up automatic competitive analysis in the new framework developed in this thesis, it shall be implemented on a GPU-architecture with CUDA.

CHAPTER 2

State of the Art

2.1 Competitive analysis of real-time scheduling

In the context of real-time scheduling, competitive analysis is a way to judge the performance of an online real-time scheduling algorithm for firm-deadline tasks. An online real-time scheduling algorithm does not know in advance which task instances are going to be released, or when they are going to be released. Therefore, it has to make all its scheduling decisions with limited information, specifically just with the task instances that have already been released at the time-slot where the decision has to be made. By contrast, an offline real-time scheduling algorithm knows all the released task instances including their release times from the start, which means it can produce a schedule which is optimal (with respect to collected utility) for the given sequence of task releases, also known as a task sequence.

Clearly, comparing the collected utility of an online algorithm with the collected utility of an optimal offline algorithm is a sound way to judge the performance of the online algorithm. The competitive ratio of an online algorithm on a particular task set is the worst case of the online algorithm's collected utility divided by the offline algorithm's collected utility, evaluated over all possible task sequences of the studied task set.

Regarding game theory and competitive analysis of scheduling algorithms, we are not aware of much work: [SBHP11] considered non-preemptive scheduling of periodic hard deadline tasks, [BMS10] used graph games for feasibility analysis in a multiprocessor environment, and [LMMW16] studies competitive synthesis with the goal of minimizing completion times.

Regarding relevant optimization techniques, [BHK17] introduced some optimizations for finding the minimum ratio cycle in a directed graph, and [BB15] might be an efficient way to use the Bellman-Ford algorithm for negative cycle detection, which could potentially be an alternative to Madani's algorithm [Mad02] for finding the minimum mean cycle.

The framework presented in [CPKS18] has laid the groundwork for this thesis. We will now describe some details of its internal workings:

Since the set of all possible task sequences of a given task set is generally infinite, some additional assumptions were made in [CPKS18] to simplify the problem of finding the worst case ratio of online utility to offline utility. If $\mathcal{T}$ is finite and only one instance of each task can be released per time-slot, then the set of currently active task instances must also be finite at all times during all executions of the scheduling algorithm, since it is bounded by $|\mathcal{T}|D_{max}$, where D_{max} is the maximum relative deadline of all the tasks in $\mathcal{T}$. Therefore, the current state of the algorithm can be defined as the set of currently active task instances together with the internal memory of the scheduling algorithm, and if the internal memory of the algorithm is finite, then the state must also be finite.

With this, the algorithm can be represented as a labeled transition system, where each node is labeled with a unique state of the algorithm, and each edge is labeled with a set of task releases. Intuitively, every edge represents the passing of one time-slot, labeled with the task instances that are released at this time-slot, and leads to the node labeled with the state the algorithm is in after the scheduling decision given the newly released task instances has been made. Also, if the scheduling algorithm collects some utility, the edge is additionally labeled with the amount of utility collected. Each infinite path in this labeled transition system corresponds to a task sequence of $\mathcal{T}$.

Since the framework [CPKS18] only considers deterministic online algorithms, the LTS of the online algorithm is always deterministic, which means that the task release labels of the outgoing edges are unique for each state. Therefore, each task sequence of $\mathcal{T}$ corresponds to precisely one infinite path in its LTS. By contrast, the LTS of the offline algorithm is non-deterministic, which intuitively means that the offline algorithm might make different scheduling decisions on the same (partial) input. This is unavoidable, because the task sequence up to a particular time-slot (corresponding to a finite path in the LTS) is not enough information for the offline algorithm to make its scheduling decision.

The labeled transition systems (or graphs for short) generated by this method can be further constrained: For example, a safety constraint might disallow task sequences where too much workload is released in too short of a time. Parts of the graph that are only reachable through paths where this constraint is violated are discarded in this case. Additional constraints include liveness constraints and limit-average constraints.

When we calculate the synchronous graph product (as defined in [CPKS18]) of the online graph, the offline graphs and the constraint graphs for all considered constraints, then we get a graph where each infinite path corresponds to a task sequence of $\mathcal{T}$ that satisfies all constraints. In this product graph, the edges are labeled with the task releases, the collected utility for the online graph and the collected utility for the offline graph. Therefore, for each finite path in the graph, we can calculate the total collected utility for the online graph divided by the total collected utility for the offline graph. Since the worst case for the limit of this ratio must be a equal to the minimum ratio cycle of online utility to offline utility in this graph, we can compute the competitive ratio by running an algorithm for finding the minimum ratio cycle.

To summarize, the framework takes a task set, an online scheduling algorithm and a set of constraints as inputs, and then executes the following steps:

- Generate a graph representing possible executions of the online algorithm on the given task set.
- Generate a graph representing possible executions of an offline algorithm on the given task set.
- Calculate the product graph of these two graphs and the constraint graphs.
- In this product graph, find the cycle with the minimum total utility for the online algorithm divided by the total utility for the offline algorithm.

The total utility for the online algorithm divided by the total utility for the offline algorithm in this cycle gives the competitive ratio of the analyzed online algorithm on this task set.

Notable optimizations in this framework include:

- The offline scheduler schedules jobs to completion or not at all.
- Non-idle scheduling optimization: As long as there are some unfinished jobs in the state of the offline scheduler, the currently executed slot must never be empty, that is the offline scheduler cannot decline to schedule a job while there are still jobs it has promised to schedule to completion.
- Admissible gap optimization: The offline scheduler will never leave a gap in the execution of a job for which there is no multiset X of tasks from $\mathcal{T}$ such that $\sum_{\tau_i \in X} C_i = l$, where l is the length of the gap. We call a gap for which such a set X exists an admissible gap.

Table 2.1 show some performance results of the framework in terms of execution time depending on graph size, taken directly from [CPKS18].

2.2 CUDA

In order to speed up automatic competitive analysis in the new framework developed in this thesis, it shall be implemented on a GPU-architecture with CUDA.

[Gui] describes the basic functionality of a GPU as an array of Streaming Multiprocessors (SMs). A CUDA compute kernel is a function that is executed by many threads in

size of the product graph (nodes)	average execution time (seconds)
823	0.04
1997	0.39
4918	10.02
1064	0.14
1653	0.66
7705	51.04
1711	2.13
3707	13.88
10040	131.83
2195	5.37
9105	142.55
16817	558.04

Table 2.1: Performance results of the framework [CPKS18]

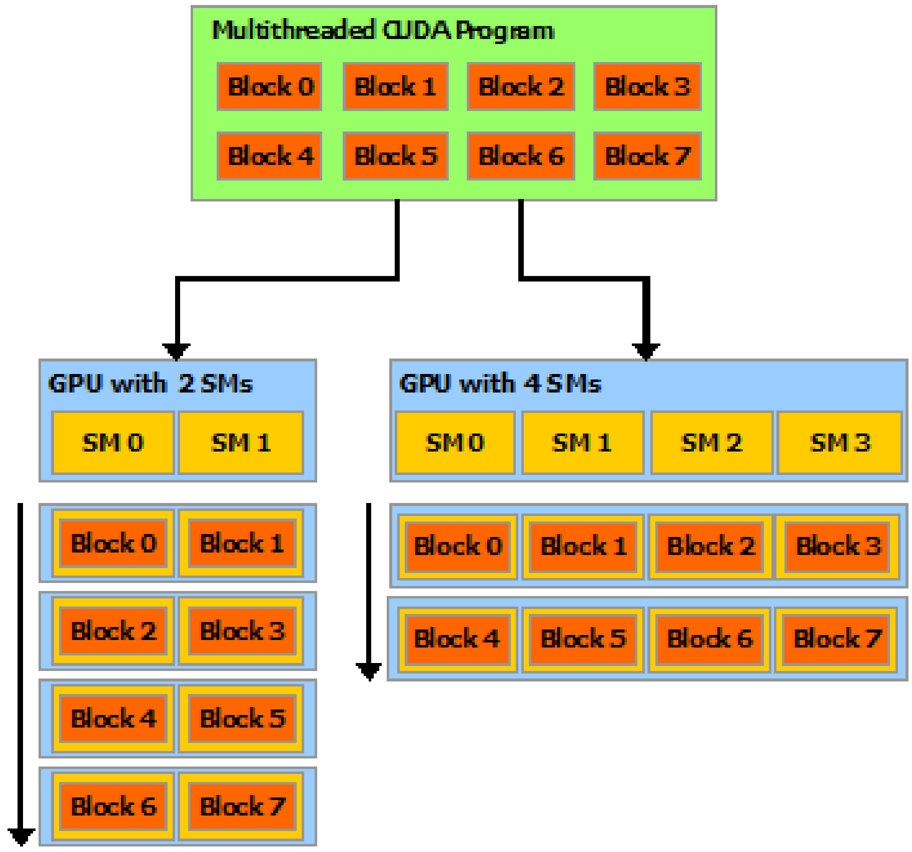

Figure 2.1: Visualization of thread blocks and SMs used in [Gui], for two different GPU-architectures (2 SMs and 4 SMs). The downward arrows represent the order of execution on each SM, with blocks that are higher up being executed first.

parallel, executing the same instructions but working on different data. These threads are organized into thread blocks that are then executed by the SMs (see Figure 2.1).

Calls to these compute kernels are made from a serial program running on the CPU. This serial code is called host code, while the code in the compute kernels is called device code. The set of all thread blocks executing the same kernel started by the same host command is called a ”grid” of threads. Per default, host code can only access host memory (RAM) and device code can only access device memory (GPU memory). This means that data transfer between host and device is necessary every time some host code wants to read memory written by device code and vice versa. Figure 2.2 illustrates an example execution, where Kernel0 is called by host code, and once the entire grid (Grid0) is finished, there is some additional host code that calls Kernel1 (corresponding to Grid1).

2.3 Graph algorithms

Aside from standard graph algorithms like depth first search [Eve79], our framework utilizes Madani’s algorithm for finding the minimum mean cycle [Mad02]. We use a reduction of the minimum ratio cycle problem to the minimum mean cycle problem utilizing guided binary search, as described in more detail in [CPKS18]. Therefore, Madani’s algorithm for finding the minimum mean cycle can be used to compute the competitive ratio, see section 7.1 for details.

Madani’s algorithm uses value iteration, which informally means that it heuristically determines the value and the outgoing edge with the smallest value for each node in each iteration. In each iteration, the value of each node is set to the minimum of edge-weight plus old target-node value among all outgoing edges of each node. If this means that a different edge now provides the minimum value for this node, then the outgoing edge with the smallest value changes as well. [Mad02] provides a proof that after n iterations, where n is the number of nodes in the graph, the history of paths in this graph consisting of only the edges with the smallest outgoing value in each iteration must contain the minimum mean cycle after at most another n iterations. The technique of searching history paths for cycles is referred to as ”super edges” in the paper. Our CUDA implementation of this algorithm is described in more detail in section 7.1.

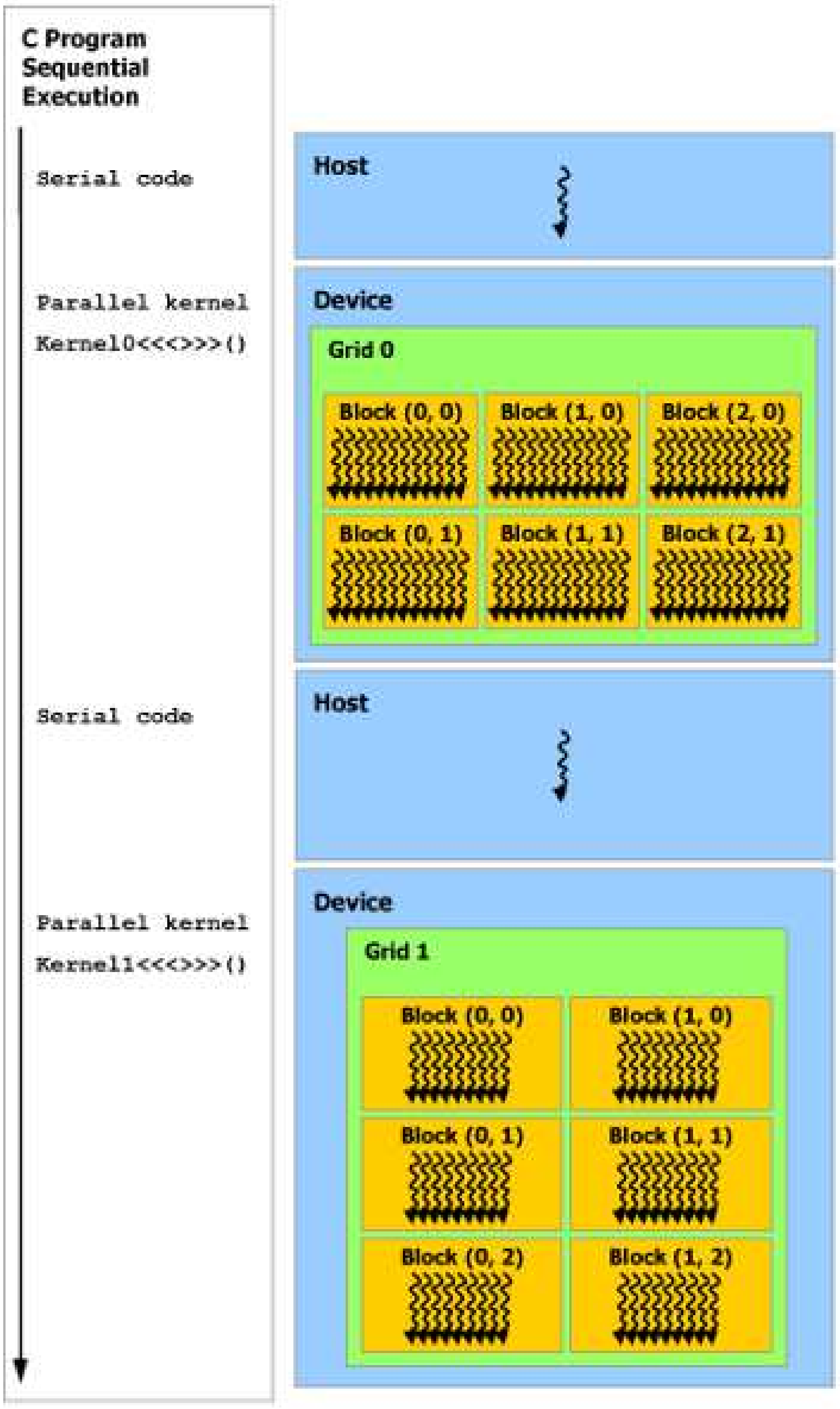

Figure 2.2: Illustration of host code and device code used in [Gui]

CHAPTER 3

Specification of the Framework

3.1 Non-preemptible sections

A non-preemptible section of a task cannot be preempted until it is completed. This means that the online algorithm does not make any scheduling decisions as long as a non-preemptible section is executed. For the offline algorithm, it means that this section can only be scheduled as a single block.

To specify a non-preemptible section, a start-timeslot and an end-timeslot within the task are needed. These timeslots are given relative to the release time of the task instance. Example: A task with a computation time of 5, and a non-preemptible section with start-timeslot 2 and end-timeslot 4 can only be preempted after the first timeslot and after the fourth timeslot. In the case of multiple non-preemptible sections within the same task, multiple pairs of start- and end-timeslots are necessary. Should the non-preemptible section span the entire rest of the task, the end-timeslot can be omitted.

3.2 Event-based precedence constraints

If Task B has an event-based dependency on Task A, then an instance of Task B is released immediately every time an instance of Task A is completed, and the adversary cannot release any instances of Task B in a different way. We will call Task A *precursor task* and Task B *dependent task*.

As an example, consider Task A with computation time 1 and relative deadline 1, and Task B with computation time 1 and relative deadline 2 (see Figure 3.2).

It is also possible to specify event-based dependencies where the dependent task is not released immediately on the completion of the precursor task, but after a constant delay.

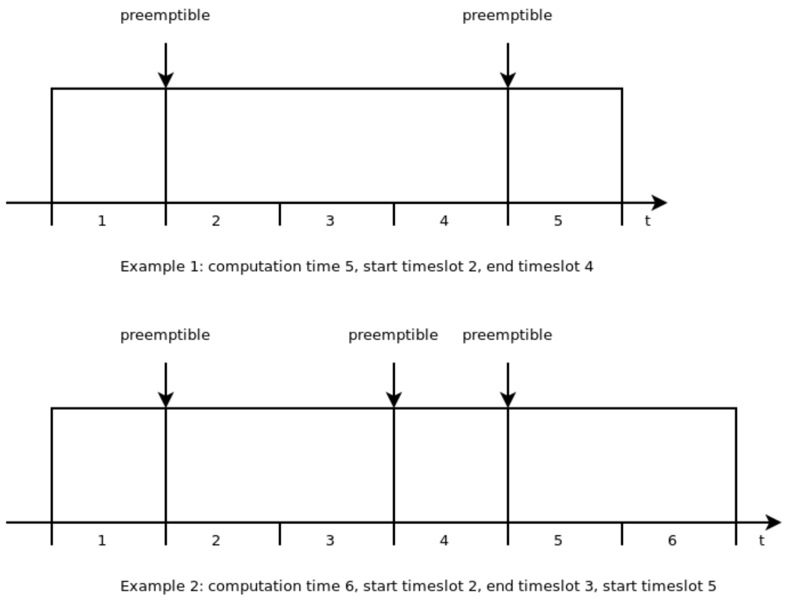
preemptible
preemptible
1
2
3
4
5
t
Example 1: computation time 5, start timeslot 2, end timeslot 4
preemptible
preemptible
preemptible
1
2
3
4
5
6
t
Example 2: computation time 6, start timeslot 2, end timeslot 3, start timeslot 5

Figure 3.1: Examples of non-preemptible sections

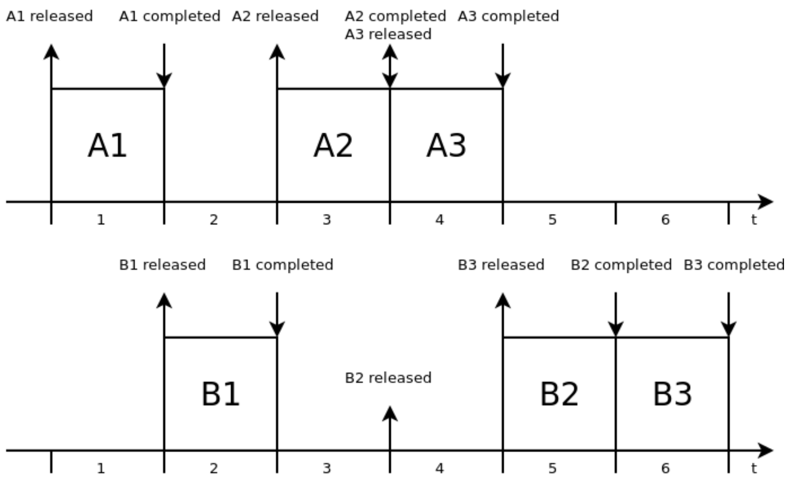

Figure 3.2: Example of an event based precedence constraint

3.3 Time-based precedence constraints

Unlike event-based dependencies, the release of time-based dependent task instances is still controlled by the adversary. If Task B has a time-based dependency on Task A, then the first released instance of Task B after an instance of Task A is completed is paired with that instance of Task A. Instances of Task B released after a paired instance of Task B and before the next instance of Task A is completed are unpaired instances. Each B in paired instances can have different values for computation time C_i', relative deadline D_i' and utility V_i' than unpaired instances, which have computation time C_i, relative deadline D_i and utility V_i. It is also possible to specify that unpaired instances of B have $C_i = D_i = V_i = 0$, in which case only paired instances of Task B are relevant to the scheduling problem, and unpaired instances are suppressed (i.e. discarded).

Consider Task A with computation time 1 and relative deadline 1, and Task B with computation time 2 and relative deadline 2. In Example 1, the computation time of Task B is reduced to 1 for paired instances. In Example 2, only paired instances of Task B are relevant, i.e. $C_i = D_i = V_i = 0$ for unpaired instances of B (see Figure 3.3).

In the general case, there is no upper bound for the time between the completion of the precursor task and the release of the paired dependent task instance. The adversary could even delay this release indefinitely. Since the release times (but not the paired/unpaired status) of time-based dependent task instances must be the same for both the online- and offline-algorithm, it is not possible to specify admissibility constraints for the time between the completion of the dependency and the release of the paired task instance. Constraints about the release times themselves (without reference to the paired/unpaired status) can be imposed normally.

3.4 Forks and joins

In a join, one dependent task has two or more precursor tasks. There are two kinds of joins: In an OR-join, the dependency is fulfilled by the completion of any precursor task instance. In an AND-join, the dependency is fulfilled by the completion of at least one instance of each precursor task. A circular dependency occurs when some task ultimately depends on itself. Unless some dependency in the dependency cycle is a time-based dependency (that only modifies the properties of the released task instance), circular dependencies need an OR-join in order to allow some task instance released by the adversary to start the cycle.

Joins are a generalization of the precedence constraints described above, in the sense that a simple precedence constraint can be thought of as a join with just one precursor task.

Forks are not explicitly modeled in our framework, but are instead created implicitly when two or more different tasks have the same precursor task. Since dependencies are only specified on the side of the dependent task, a single task can be the precursor task for an arbitrary mixture of event-based or time-based dependencies with AND- or OR-joins.

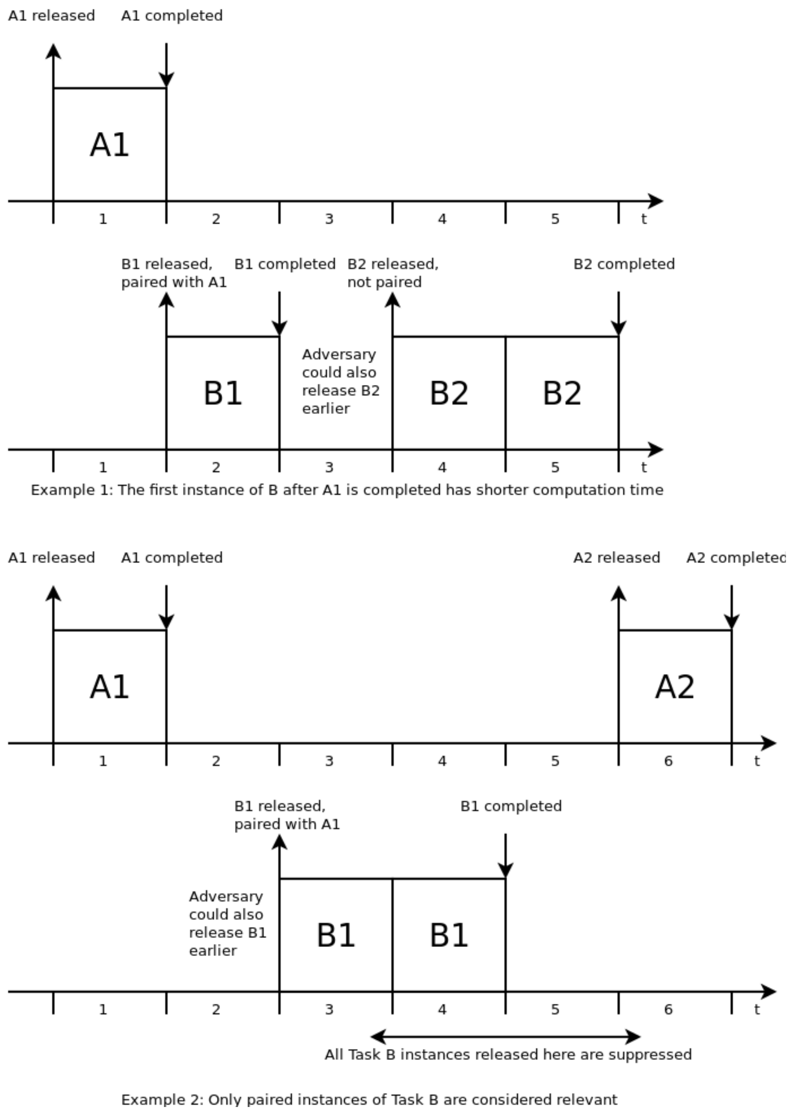
A1 released
A1 completed
A1
1
2
3
4
5
t
B1 released, paired with A1
B1 completed
B2 released, not paired
B2 completed
Adversary could also release B2 earlier
B1
B2
B2
Example 1: The first instance of B after A1 is completed has shorter computation time
A1 released
A1 completed
A2 released
A2 completed
A1
A2
6
B1 released, paired with A1
B1 completed
Adversary could also release B1 earlier
B1
B1
All Task B instances released here are suppressed
Example 2: Only paired instances of Task B are considered relevant

Figure 3.3: Examples of a time based precedence constraint

Consider Task A, B, C and D, all with computation time 1 and relative deadline 2. Task B and C have a dependency on Task A, and Task D has an AND-join dependency on B AND C. Both event-based and time-based dependencies are possible, but this example uses event-based dependencies for simplicity (see Figure 3.4).

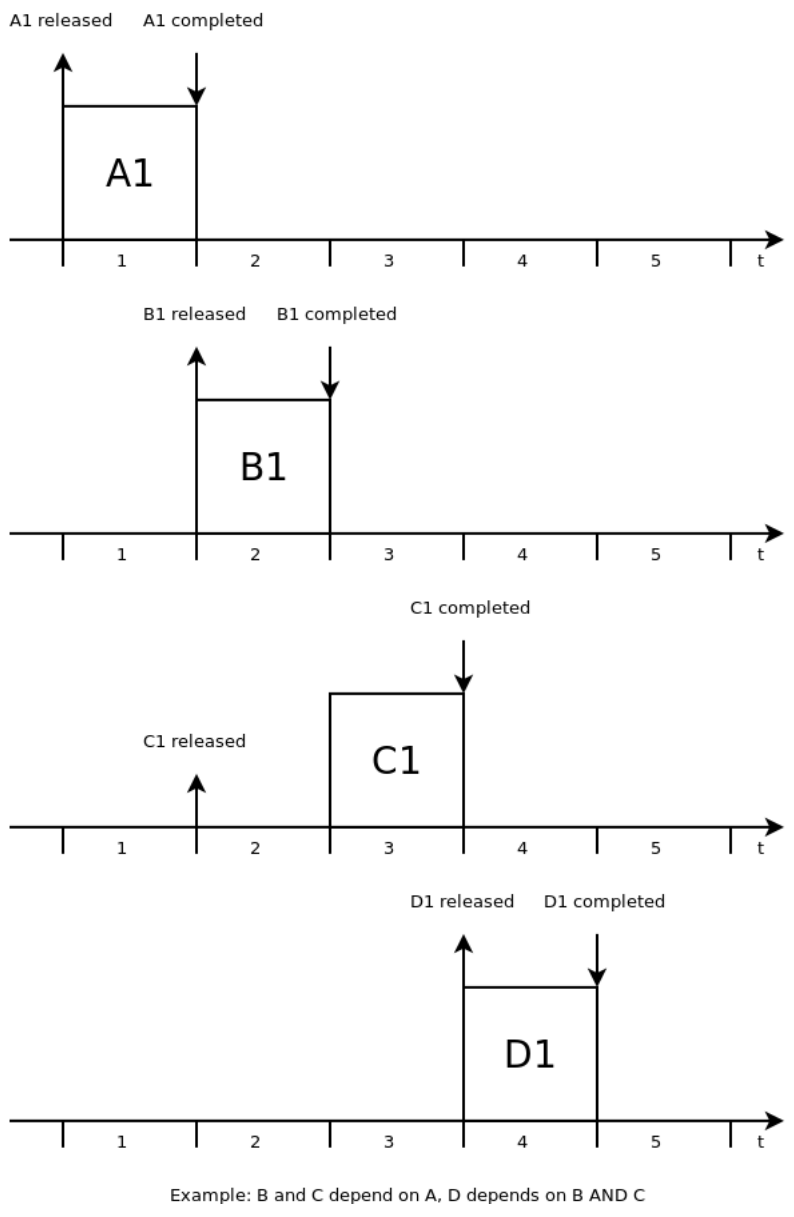

Figure 3.4: Examples of a fork followed by an AND-join

CHAPTER 4

Competitive Ratio Computation

In [CPKS18], the competitive ratio of an online algorithm A on a given set of admissible job sequences J was defined as

$$CR_J(A) = \inf_{\sigma \in J} \lim_{k \to \infty} \inf \frac{1 + V(\pi_A^\sigma, k)}{1 + V(\pi_C^\sigma, k)}, \tag{4.1}$$

with π_A^σ being the schedule of the online algorithm A, and π_C^σ being the schedule of a clairvoyant offline algorithm C working on the same job sequence σ. The term $V(\pi, k)$ is the value achieved by schedule π up to step k. In our setting, both event-based and time-based precedence constraints may result in different job sequences for A and C: A precursor task instance may be completed by C but not by A (or vice versa), resulting in an event-based dependent task instance being released for C but not for A, or a newly released time-based dependent task instance being paired for C but not for A. The definition of $CR_J(A)$ given in equation (1) is therefore not directly applicable to our model.

In order to account for this, we no longer demand that both the online and the offline algorithm work on the same job sequence σ. Instead, the online algorithm works on the job sequence σ_A and the offline algorithm works on a possibly different job sequence σ_C. Based on the notation used in [CPKS18], we define the following terms:

- The taskset $\mathcal{T} = \{\tau_1, ..., \tau_N\} = \mathcal{T}_{ed} \cup \mathcal{T}_{td} \cup \mathcal{T}_n$, where $\mathcal{T}_{ed}$ is the set of event-based dependent tasks (contains all dependent tasks in some event-based dependency), $\mathcal{T}_{td}$ is the set of time-based dependent tasks (contains all dependent tasks in some time-based dependency), and $\mathcal{T}_n$ is the set of non-dependent tasks. Each task $\tau_i = (i, C_i, D_i, V_i)$ is still characterized by the 4-tuple of task id, worst-case execution time, relative deadline and utility value. Each $\tau_i \in \mathcal{T}_{td}$ has the values of the unpaired version of the task. Our model does not allow for tasks that are dependent tasks of more than one dependency, which means that $\mathcal{T}_{ed}$, $\mathcal{T}_{td}$ and $\mathcal{T}_n$ are disjoint.

- An alternative taskset $\mathcal{T}'_{td}$, with $\tau'_i = (i, C'_i, D'_i, V'_i)$ of the paired version τ'_i of the task $\tau_i \in \mathcal{T}_{td}$. The bijective function $paired : \mathcal{T}_{td} \to \mathcal{T}'_{td}$ returns the paired version of each unpaired task: $\tau'_i = paired(\tau_i)$. This alternative taskset is necessary because different schedules might result in the same task release being a paired or unpaired instance.

- The set of possible non-dependent task releases per slot $\Theta_n = \{\theta_n | \theta_n \in 2^{\mathcal{T}_n}\}$.

- The set of all non-dependent task release sequences $\Sigma_n = \Theta_n^\omega$, and the set of all non-dependent task release sequence prefixes of length t, $\Sigma_n(t) = \Theta_n^t$. For $\sigma_n \in \Sigma_n$, $\sigma_n(t) = (\sigma_n^1, \sigma_n^2, ..., \sigma_n^t)$ is the t-prefix of σ_n, with σ_n^t being the non-dependent task releases in slot t.

- The set of possible anonymous time-dependent task releases per slot $\Theta_r = \{\theta_r | \theta_r \in 2^{\mathcal{T}_{td}}\}$. Note that "anonymous" in this context means that it is not yet decided whether or not this task instance will be paired or unpaired. The unpaired instances in this set might be converted to paired instances depending on context.

- The set of all anonymous time-dependent task release sequences $\Sigma_r = \Theta_r^\omega$, and the set of all anonymous time-dependent task release sequence prefixes of length t, $\Sigma_r(t) = \Theta_r^t$. For $\sigma_r \in \Sigma_r$, $\sigma_r(t) = (\sigma_r^1, \sigma_r^2, ..., \sigma_r^t)$ is the t-prefix of σ_r, with σ_r^t being the anonymous time-dependent task releases in slot t.

- The set of all schedules $\Pi = (\{(\mathcal{T} \cup \mathcal{T}'_{td}) \times \{0, ..., D_{max} - 1\}\} \cup \{\emptyset\})^\omega$, and the set of all schedule prefixes of length t, $\Pi(t) = (\{(\mathcal{T} \cup \mathcal{T}'_{td}) \times \{0, ..., D_{max} - 1\}\} \cup \{\emptyset\})^t$. For $\pi \in \Pi$, $\pi(t) = (\pi^1, \pi^2, ..., \pi^t)$ is the t-prefix of π, with π^t being scheduled in slot t with remaining relative deadline.

- The set of admissible sequences of releases of non-dependent and anonymous time-dependent tasks $J = \{\sigma_n \cup \sigma_r | \sigma_n \in \Sigma_n, \sigma_r \in \Sigma_r\}$, where $\sigma_n \cup \sigma_r$ means elementwise union. This set can be subjected to additional constraints, as described in [CPKS18].

Unfortunately, because of event- and time-based dependencies, just knowing $\sigma_n \cup \sigma_r$ is not sufficient for completely specifying the schedule π_A of the online-algorithm or the schedule π_C of the offline algorithm respectively, as they face two different task release sequences σ_A and σ_C for the very same $\sigma_n \cup \sigma_r$. We therefore need additional definitions:

- The set of possible task releases per slot $\Theta = \{\theta | \theta \in 2^{\mathcal{T} \cup \mathcal{T}'_{td}}$, with $\tau_i \in \theta \Rightarrow paired(\tau_i) \notin \theta\}$ for all $\tau_i \in \mathcal{T}_{td}$.

- The set of all task release sequences $\Sigma = \Theta^\omega$, and the set of all task release sequence prefixes of length t, $\Sigma(t) = \Theta^t$. For $\sigma \in \Sigma$, $\sigma(t) = (\sigma^1, \sigma^2, ..., \sigma^t)$ is the t-prefix of σ, with σ^t being the task releases in slot t.

- $\sigma_A \in \Sigma$ is the job sequence for algorithm A, and $\sigma_C \in \Sigma$ is the job sequence for algorithm C.

Given σ_A, $\pi_A^{\sigma_A}$ is the schedule for σ_A produced by algorithm A. Note that $\pi_A^{\sigma_A}(t)$ only depends on $\sigma_A(t)$, since A is an online algorithm. However: $\sigma_A(t)$ also depends on $\pi_A^{\sigma_A}(t-1)$, since σ_A^t depends on it. This dependency is not circular, since $\pi_A^{\sigma_A}(t-1)$ only depends on $\sigma_A(t-1)$. In order to account for this, we introduce the concept of release functions:

The event-based release function E_τ^t is a function parameterized by $\tau \in \mathcal{T}_{ed}$ and $t \in \mathbb{N}$, and the signature is $E_\tau^t : \Pi(t-1) \to \{\emptyset, \tau\}$. For any event-based dependent task $\tau \in \mathcal{T}_{ed}$ and any schedule prefix $\pi(t-1)$, $E_\tau^t(\pi(t-1))$ determines whether a job of task τ is released at time t or not, depending on the schedule prefix up to time $t-1$. Formally,

$$E_\tau^t(\pi(t-1)) = \begin{cases} \tau, & \text{if } \pi(t-1) \text{ demands an instance of } \tau \text{ to be released at time } t, \\ \emptyset, & \text{otherwise.} \end{cases} \tag{4.2}$$

Informally, this function returns τ if $\pi(t-1)$ completed the dependency of τ at time $t-1$ and $\emptyset$ otherwise. Note that the exact behavior of this function could potentially be generalized to support arbitrary conditions on $\pi(t-1)$. Since the relative deadline and worst case execution time of each scheduled task is encoded in $\pi(t-1)$, it is sufficient to determine the completion time of precursor task instances. Our framework defines the exact interpretation of "demands an instance of τ to be released" by a labeled transition system, which is presented later. In order to get all the event-based dependent task releases at time t, we simply look at the union of the release functions (recall that, in our context, union notation on sequences just means the elementwise union):

$$E^t(\pi(t-1)) = \bigcup_{\tau \in \mathcal{T}_{ed}} E_\tau^t(\pi(t-1)). \tag{4.3}$$

The time-based release function T_τ^t is a function parameterized by $\tau \in \mathcal{T}_{td}$ and $t \in \mathbb{N}$, and the signature is $T_\tau^t : \Theta_r, \Sigma(t-1), \Pi(t-1) \to \{\emptyset, \tau, paired(\tau)\}$. For any time-based dependent task $\tau \in \mathcal{T}_{td}$ with an anonymous task release $\tau \in \theta_r^t$ at time t, any schedule prefix $\pi(t-1)$ and any job release prefix $\sigma(t-1)$, the release function $T_\tau^t(\theta_r^t, \sigma(t-1), \pi(t-1))$ returns either $\emptyset$, τ or $paired(\tau)$, depending on whether $\tau \in \theta_r^t$ and on whether its dependency is fulfilled or not. Formally,

$$T^t_\tau(\theta^t_r, \sigma(t-1), \pi(t-1)) = \begin{cases} paired(\tau), & \text{if } \tau \in \theta^t_r \text{ and } \pi(t-1) \text{ demands a paired instance,} \\ \tau, & \text{if } \tau \in \theta^t_r \text{ and } \pi(t-1) \text{ does not demand a paired instance,} \\ \emptyset, & \text{if } \tau \notin \theta^t_r. \end{cases} \tag{4.4}$$

Informally, this function converts the unpaired instance of τ into a paired instance if $\pi(t-1)$ contains the completion of its dependency after the last release of τ in $\sigma(t-1)$. Again, the exact behavior of this function could be generalized to other conditions. In order to get all time-based dependent task releases at time t, we again take the union of release functions:

$$T^t(\theta^t_r, \sigma(t-1), \pi(t-1)) = \bigcup_{\tau \in \mathcal{T}_{td}} T^t_\tau(\theta^t_r, \sigma(t-1), \pi(t-1)). \tag{4.5}$$

With this, we can define the job sequence $\sigma_A = \sigma_{A,n} \cup \sigma_{A,ed} \cup \sigma_{A,td}$ for the online algorithm A as the union of the following subsequences (except $\sigma_{A,r}$, which is just used to construct $\sigma_{A,td}$):

- $\sigma_{A,n}$: The non-dependent task releases, a priori given.
- $\sigma_{A,r}$: All anonymous task releases of time-based dependent tasks $\tau \in \mathcal{T}_{td}$, a priori given (but no distinction between paired or unpaired instances).
- $\sigma_{A,ed}$: Event-based dependent task releases, defined inductively for $t \geq 1$: $\sigma^1_{A,ed} = \emptyset$, $\sigma^t_{A,ed} = E^t(\pi_A(t-1))$, where $\pi_A(t-1)$ is the schedule for $\sigma_A(t-1)$.
- $\sigma_{A,td}$: Time-based dependent task releases, defined inductively for $t \geq 1$: $\sigma^1_{A,td} = T^1(\sigma^1_{A,r}, \emptyset, \emptyset)$, $\sigma^t_{A,td} = T^t(\sigma^t_{A,r}, \sigma_A(t-1), \pi_A(t-1))$, where $\pi_A(t-1)$ is the schedule for $\sigma_A(t-1)$.

Note that the adversary only has control over $\sigma_{A,n} \cup \sigma_{A,r}$. $\sigma_{A,ed}$ and $\sigma_{A,td}$ are then uniquely determined by this and the schedule produced by the algorithm.

For the offline algorithm C, the situation is slightly more complicated, since there is no unique schedule prefix $\pi_C(t)$ for each finite job sequence prefix $\sigma_C(t)$. However, there is still a finite set of possible schedule prefixes $\Pi_C(t)$ given a finite job sequence prefix $\sigma_C(t)$, and this set remains unchanged regardless of how σ_C is extended after timeslot t. Therefore, if we treat $\sigma_{C,n}$ and $\sigma_{C,r}$ as a priori given, as we did for the online algorithm, we can inductively define the set of possible schedules $\Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}$ and the set of possible job sequences $\Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}$ together:

1. The set of possible job releases at $t = 1$ is $\Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(1) = \sigma_{C,n}^1 \cup T^1(\sigma_{C,r}^1, \emptyset, \emptyset)$, i.e. just the non-dependent task releases and time-based dependent task releases (that are guaranteed to be unpaired).

2. The set of possible schedules at $t = 1$ is $\Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(1) = \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(1) \cup \{\emptyset\}$, i.e., the offline algorithm can schedule one of the tasks that have been released at time $t = 1$ or nothing.

3. For each matching task release sequence prefix $\sigma_C(t-1) \in \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t-1)$ and schedule prefix $\pi_C(t-1) \in \Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t-1)$, the task release sequence prefix $\sigma_C(t)$, which is $\sigma_C(t-1)$ extended by $\sigma_C^t = \sigma_{C,n}^t \cup E^t(\pi_C(t-1)) \cup T^t(\sigma_{C,r}^t, \sigma_C(t-1), \pi_C(t-1))$, must be in $\Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t)$.

4. For each task release sequence prefix $\sigma_C(t) \in \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t)$ generated in step 3 by extending the matching task release sequence prefix $\sigma_C(t-1) \in \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t-1)$ and schedule prefix $\pi_C(t-1) \in \Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t-1)$, all schedule prefixes $\pi_C(t)$, which are $\pi_C(t-1)$ extended by the scheduling of an active job given $\sigma_C(t)$ and $\pi_C(t-1)$ or no job, must be in $\Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t)$.

This way, $\Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t)$ and $\Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t)$ can be constructed for arbitrarily large t. Therefore, we can take the limit to extend the prefixes to infinite sequences:

$$\lim_{t \to \infty} \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t) = \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})} \tag{4.6}$$

$$\lim_{t \to \infty} \Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}(t) = \Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})} \tag{4.7}$$

For any job sequence $\sigma_C \in \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}$, there exists a schedule $\pi_C \in \Pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}$ that fulfills the constraints of event-based and time-based dependencies by construction. For any given $\sigma_{C,n} \cup \sigma_{C,r}$, each possible schedule $\sigma_C \in \Sigma_C^{(\sigma_{C,n} \cup \sigma_{C,r})}$ implicitly relies on unique sequences $\sigma_{C,ed}$ and $\sigma_{C,td}$. The offline algorithm can therefore just select the optimal schedule $\pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}$ among all the possible schedules given $\sigma_{C,n}$ and $\sigma_{C,r}$, because $\pi_C^{(\sigma_{C,n} \cup \sigma_{C,r})}$ uniquely determines $\sigma_{C,ed}$ and $\sigma_{C,td}$.

In order for the competitive ratio to make sense, we assume that both the online- and the offline algorithm work on the same sequence of non-dependent and anonymous time-dependent task releases $\sigma_n \cup \sigma_r$. Therefore, $\sigma_{A,n} = \sigma_{C,n} = \sigma_n$ and $\sigma_{A,r} = \sigma_{C,r} = \sigma_r$. $\pi_A^{(\sigma_n \cup \sigma_r)}$ refers to the schedule produced by the online algorithm A given $\sigma_n \cup \sigma_r$, with event-based and time-based dependent task releases being incorporated into the job sequence as described above. The same goes for $\pi_C^{(\sigma_n \cup \sigma_r)}$ and the offline algorithm C.

With this, given a certain taskset including its subsets, release functions and other constraints, we can rewrite the definition of the competitive ratio as follows to fit our model:

$$CR_J(A) = \inf_{(\sigma_n \cup \sigma_r) \in J} \lim_{k \to \infty} \inf \frac{1 + V(\pi_A^{(\sigma_n \cup \sigma_r)}, k)}{1 + V(\pi_C^{(\sigma_n \cup \sigma_r)}, k)} \tag{4.8}$$

where J is the set of admissible sequences of releases of non-dependent and anonymous time-dependent tasks $(\sigma_n \cup \sigma_r)$. Note that in contrast to the original model, J is just a subset of the full job sequences the online- and offline-algorithm work on, but as explained above, J together with A or C uniquely determines both the schedule and the rest of the job sequences (which may be different between A and C).

4.1 Release functions as labeled transition systems

Previously, we defined release functions for "demanding" the release of an event-based dependent task, or a paired version of a time-based dependent task. In our framework, this can be implemented by labeled transition systems, which fall into four categories (see Figure 4.1 for an illustration):

- Event-based OR-dependencies: There is a state labeled "uncompleted" and a state labeled "completed". For each precursor task, there is an edge labeled with the task completion from the uncompleted state to the completed state. There is a single edge labeled with the dependent task release from the completed state to the uncompleted state.
- Time-based OR-dependencies: The LTS is identical to the one for event-based OR-dependencies, except that the uncompleted state has a self-loop labeled with the unpaired version of the dependent task release, and the edge from the completed state to the uncompleted state is labeled with the paired version of the dependent task release.
- Event-based AND-dependencies: There are 2^n states, where n is the number of precursor tasks, and each state is labeled with a different subset of the set of precursor tasks. If s_1 and s_2 are states, and s_1 is labeled with a subset of the label of s_2 such that exactly one precursor task in the s_2 label is missing from the s_1 label, then there is an edge from s_1 to s_2 labeled with the completion of the missing precursor task. Additionally, there is an edge labeled with the dependent task release from the state labeled with the entire set of precursor tasks to the state labeled with the empty set.
- Time-based AND-dependencies: The LTS is again identical to the one for event-based AND-dependencies, except that the edge from the state labeled with the

entire set of precursor tasks to the state labeled with the empty set is labeled with the paired version of the dependent task release, and each state that is not labeled with the entire set of precursor tasks has a self-loop labeled with the unpaired version of the dependent task release.

As an example, consider the case where task C depends on the tasks A and B (see Figure 4.1).

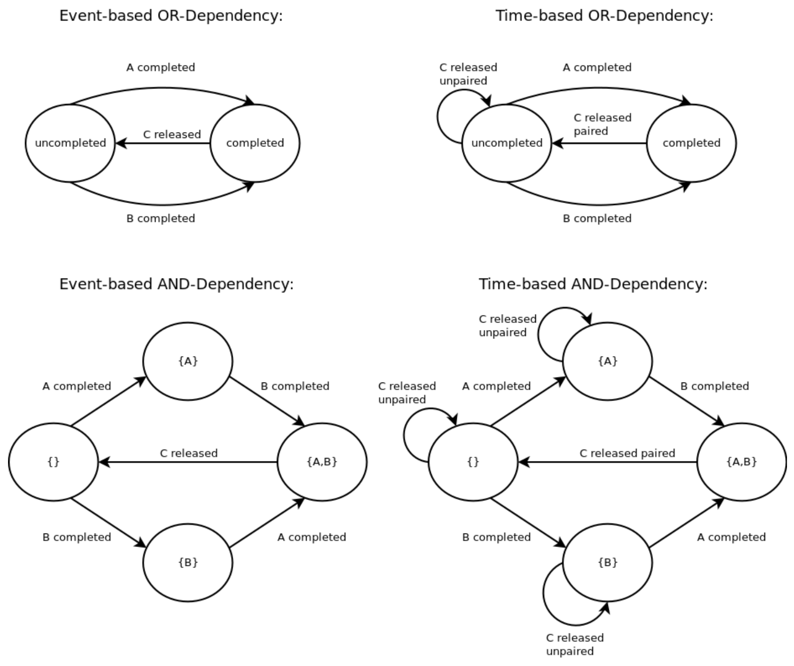

Figure 4.1: Examples of release function LTSs

CHAPTER 5

Graph Generation

In the online graphs of our framework, nodes correspond to states of the scheduler and edges correspond to possible task releases. The state of the scheduler includes currently active task instances with their remaining workload and relative deadline, as well as which tasks have currently been completed for each dependency. Certain algorithms also require additional information to be saved in the state. Each node must have an outgoing edge for each possible combination of non-dependent and anonymous time-based dependent task releases. Event-based dependent task releases and the paired or unpaired status of time-based dependent task releases are determined by the state.

The online graph is generated by starting with a node corresponding to the initial state, and then recursively adding the required outgoing edges and the nodes they lead to into the graph. The internal state of each newly added node is calculated by taking the state of the previous node, updating the relative deadlines, adding all the released tasks and then applying the scheduling decision of the online algorithm. For this purpose, each online algorithm has a function for choosing the scheduled task instance and then reducing the state, which are given in Appendix A. Should the state of the new node be equal to the state of an already existing node, they are merged.

For the offline algorithm, nodes also correspond to states, but edges correspond not only to task releases, but also to possible scheduling decisions of the offline algorithm. The offline algorithm reserves future time-slots for task-instances it intends to complete ahead of time, which means the state must contain information about which time-slots have already been reserved in this way. The latest possible point at which a task instance can be completed is its release time plus its relative deadline D_i. Since scheduling decisions are usually made at the release-time of the task, a bit-vector with length D_{max} is usually sufficient to represent the information of which future time-slots have already been reserved. However, there is one exception: The scheduling decision for an event-based dependent task with a non-zero release delay is not made at release time, i.e. the time of the completion of the last required precursor task, but release delay slots later. This means that the required length of the bit-vector is the maximum of D_{max} and the largest sum of relative deadline and release delay among event-based dependent tasks with non-zero release delay. The information for the dependencies, which also includes task instances that are already scheduled to be completed, is also stored in the state. Each node must have an outgoing edge not only for each possible combination of non-dependent and anonymous time-based dependent task releases, but also for each possible scheduling decision given those releases, except the ones that cannot possibly be optimal. The offline graph is then generated in the same manner as the online graph.

Once both the offline graph and the online graph are generated, their graph product is calculated. In the resulting graph, each node corresponds to an online-state and an offline-state, and each edge corresponds to a combination of non-dependent and anonymous time-based dependent task releases and a scheduling-decision by the offline algorithm. Each of these edges can be labeled with the utility that the online algorithm and the offline algorithm would receive given the task releases determined by the edge.

5.1 Online algorithms

The following online algorithms are implemented in the framework:

- Earliest deadline first: Schedules the job with the earliest absolute deadline. Ties are broken in favor of the job with the smallest task-index.
- First in, first out: Schedules the job with the earliest release time. Ties are broken in favor of the job with the smallest task-index.
- Static priorities: Schedules the job with the highest static priority. Ties are broken in favor of the job with the earliest release time.
- Smallest remaining time: Schedules the job with the smallest workload remaining. Ties are broken in favor of the job with the earliest absolute deadline, and then in favor of the job with the smallest task-index.
- Profit Density: This algorithm is based on [ZWL13], and schedules the job with the highest profit density, where profit density is defined as the utility of the task divided by the remaining workload of the job. This algorithm has also been made precedence-aware, in the sense that the profit density of a precursor task is increased in case a dependent task with a currently unfulfilled dependency has a higher profit density. Specifically, if v_d is the value of the dependent task, v_p the value of the precursor task, c_d the workload of the dependent task and c_p the remaining workload of the precursor task instance considered for scheduling, then the profit density of this instance is $max(\frac{v_p}{c_p}, \frac{v_p+v_d}{c_p+c_d})$. Ties are broken in favor of the job with the earliest absolute deadline, and then in favor of the job with the smallest task-index.

- Smallest slack time [SAr+04]: Schedules the job with the smallest slack time (laxity), where slack time is defined as relative deadline minus remaining workload. Ties are broken in favor of the job with the smallest task-index, and then in favor of the job with the earliest release time.
- Dover: This algorithm is described in [KS95]. It behaves like earliest deadline first in underloaded conditions and has special logic for overload.
- Dstar: This algorithm is described in [BKM+91]. Like Dover, it behaves like earliest deadline first in underloaded conditions and has special logic for overload.

The implementation of these algorithms in C is included in Appendix A.

5.2 Formal description of the offline algorithm

The state s of the offline algorithm at a particular time-slot consists of the following information:

- For the current time-slot as well as future time-slots, has the offline algorithm already promised to schedule some task at this time-slot? This is expressed as a function $p_s : \mathbb{N} \to \{true, false\}$, where $p_s(n)$ is true if and only if in the state s, the offline algorithm has already promised to schedule some task in the time-slot that is n time-slots into the future ($n = 0$ means the current time-slot).
- What is the current state of each dependency LTS? For each dependent task τ, $d_{s\tau}$ is the set of precursor tasks of τ that has been completed since the last release of τ in the state s. $er_s : \tau \to \{true, false\}$ is a function that returns true if and only if τ has an event-based dependency and the LTS of $d_{s\tau}$ is in a state where an instance of τ can be released. $tr_s : \tau \to \{true, false\}$ is a function that returns true if and only if τ has a time-based dependency and the LTS of $d_{s\tau}$ is in a state where a paired instance of τ can be released.
- Which already released task instances will be completed in the future? This is expressed as a function $c_s : \mathbb{N} \to \{\tau_{null} \cup \mathcal{T}\}$, where $c_s(n)$ gives the task τ of an already released task instance that will be completed n time-slots in the future in state s, or τ_{null} if no already released task instance will be completed n time-slots in the future in state s.

The initial state s_0 is defined as the state where $p_{s_0}(n) = false$, for all $n \in \mathbb{N}$, $d_{s_0\tau} = \emptyset$ for all $\tau \in \mathcal{T}$, and $c_{s_0}(n) = \tau_{null}$ for all $n \in \mathbb{N}$.

Transitions from a state s_1 to another state s_2 of the offline algorithm fall into three categories:

- The scheduling of an instance of task τ_i by the offline algorithm (in short q_i-transition), where there is a set $Q \subseteq \{0, 1, ..., D_i\}$ with $|Q| = C_i$ and $\forall n \in Q : p_{s_1}(n) = false, p_{s_2}(n) = true$. For non-preemptible sections, there is an additional constraint that the corresponding numbers in Q must be consecutive. Also, if τ_i is a precursor task for any dependency in the studied task set, then for the largest number $n \in Q$, it must be the case that $c_{s_2}(n) = \tau_i$. For tasks that are not precursor tasks of any dependency, it is assumed that $c_{s_2}(n) = \tau_{null}$ for $n \in Q$, because information about when exactly these tasks will be completed has no further impact on scheduling decisions or task releases, and is not necessary for computing the competitive ratio. If τ_i is an event-based dependent task with a non-zero release delay e, then numbers $n \in Q$ must be in the range $\{e, e+1, ..., e+D_i\}$ instead of $\{0, 1, ..., D_i\}$.

- Advancing one time-slot (in short a-transition), where $p_{s_2}(n) = p_{s_1}(n+1)$, $c_{s_2}(n) = c_{s_1}(n+1)$, and if $c_{s_1}(0) = \tau_i$ then $\tau_i \in d_{s_2\tau}$ for all dependent tasks τ for which τ_i is a precursor task.

- Releasing a dependent task τ_i (in short d_i-transition), where either $er_{s_1}(\tau_i) = true$ or $tr_{s_1}(\tau_i) = true$ and $d_{s_2\tau_i} = \emptyset$.

A single edge in the offline graph from a state s_1 to another state s_2 consists of the following state-transitions:

- First, a number of d_i-transitions in ascending order of i. Since the order of d_i-transitions does not matter, this choice of ordering them is arbitrary. If $er_{s_1}(\tau_i) = true$ then every edge starting in s_1 must contain a d_i transition for τ_i. This is to ensure that tasks with an event-based dependency are actually released once their dependency is fulfilled.

- Second, a number of q_i-transitions ordered by the priority of the associated tasks τ_i, where the priority of a task usually corresponds inversely with its relative deadline, and with higher priority tasks occuring before lower priority tasks. In each edge there can be at most one q_i-transitions for each task τ_i, which ensures that at most one job of each task is released in a single time-slot. q_i-transitions for event-based dependent tasks are only allowed to occur in an edge if the same edge contains a d_i-transition for the same task τ_i. q_i-transitions for time-based dependent tasks represent the release of a paired instance of τ_i if the same edge contains a corresponding d_i-transition, or an unpaired instance of τ_i if it does not. q_i-transitions for time-based dependent tasks are not allowed if $tr_{s_1}(\tau_i) = true$ unless there was a d_i-transition in the same edge, which forces time-based dependent task releases to be paired if their dependencies are met.

- Third, an a-transition, which is the final transitions of each edge and advances the state of the offline algorithm to the next time-slot.

Since the structure of q_i-transitions ensures that all jobs that are nondeterministically chosen to be scheduled at all must be scheduled to completion, the offline algorithm will never be overloaded. EDF* [CSB90] [Bla77] is optimal in underloaded conditions even in the presence of precedence relations, so we can assign the priorities of tasks according to EDF*:

The priority-order of tasks τ_i corresponds to their relative deadline D_i, with tasks with lower D_i having higher priority. Precursor-tasks have a place in the priority-order determined by their own D_i, just like every other task. However, if $D_j < D_i$ for some dependent task τ_j that has a dependency on τ_i, then τ_i may also appear as if its own relative deadline were equal to D_j. For dependent tasks τ_j that are themselves precursor tasks, the value of D_j can itself have multiple possibilities that might be optimal, and each one of these possibilities must be accounted for when calculating the possible places in the priority order for a task with transitive dependencies. This higher priority is only warranted if the dependent task τ_j is actually released and scheduled, though, so the offline algorithm chooses nondeterministically between all the possible positions in the priority list for each precursor-task, and generated a corresponding edge in the offline graph, in order to cover all possibilities for future dependent task releases. Altough a precursor-task τ_i may thus occupy several different positions in the priority-list, it may still only be released at most once each time-slot.

With EDF* being optimal in underloaded conditions, and all q_i-transitions of each edge ordered by their priority, we can impose an additional constraint on the q_i-transitions in each edge of the offline graph:

Lemma 1. *The smallest number $q \in Q$ of each q_i transition must be larger than the largest number $q' \in Q$ of the previous q_i-transition in the same offline-edge.*

This ensures that for jobs released at the same time, a job with lower priority cannot be scheduled before a job with higher priority.

An execution path of the offline algorithm is a sequence edges of the offline graph, and has the following properties:

- The non-dependent and anonymous time-based dependent task releases of each edge are given by the set of non-dependent or time-based dependent tasks for which a q_i-transition took place in that edge. The sequence of these releases is a property of the execution path.
- The utility of an execution path is given by the sum of the corresponding V_i for each q_i-transition on the path. In the case where τ_i has a time-based dependency, it must be taken into account whether or not the q_i-transition corresponds to a paired or unpaired task release, which can be determined by looking at the d_i-transitions in the same edge.

CHAPTER 6

Scheduling Anomalies

Non-preemptible sections and precedence constraints can result in some situations where the optimizations for the offline algorithm used in [CPKS18] are no longer straightforwardly valid. For example, previously it could never be optimal for the offline algorithm to not schedule a job immediately if the current slot is still empty and the job is scheduled at all. With the new model, the non-idling optimization is no longer optimal in some situations. Here are some examples:

In the case of nonpreemptible sections, consider the release of a job with $C_i = 2, D_i = 4$ and a nonpreemptible section spanning the entire job. If the offline algorithm is currently in a state where the first, third and fourth slots are free, and the second slot already contains a nondeterministically scheduled block from a different job, then it is optimal to schedule this job in the third and fourth slots and not schedule anything in the first slot, since the nonpreemptible job does not fit in the first slot. Note that the offline algorithm nondeterministically tries all possible ways to schedule a job, as long as they are not guaranteed to be non-optimal, so this initial state is in fact reachable given the right taskset.

In the case of event-based dependencies, consider a dependent task with $C_i = D_i = 2$ and a precursor task with $C_i = 1, D_i = 3$, and the same initial state as in the previous example. If an instance of the precursor task is released, then it can only be scheduled in the first slot or the third slot; If it was scheduled in the first slot, then a dependent task instance is released immediately afterwards, but cannot be scheduled, since the second slot is already occupied. If there are no other task releases in the meantime, it is therefore optimal to schedule the precursor task instance in the third slot, since the fourth and fifth slots are unoccupied and the dependent task instance can therefore be scheduled.

In the case of time-based dependencies, in cases where the unpaired task has more advantageous (i.e., ones that make scheduling easier) properties than the paired task, it can be optimal to delay the completion of the precursor task for similar reasons as in the event-based example. Assume again the same initial state as before, and between the first and second slot, an instance of the time-based task is released, and we again have a precursor task with $C_i = 1, D_i = 3$. Assuming no precursor task instance has been completed since the last dependent task release, the time-based task instance will be paired if the precursor task instance is scheduled in the first slot, or unpaired if it is scheduled in the third slot. If the paired instance of the dependent task has $C_i = D_i = 1$, and the unpaired instance has $C_i = 1, D_i = 3$, then it is again optimal to schedule the precursor task in the third slot instead of the first, since the paired version cannot be scheduled but the unpaired version can be.

6.1 Preserving the optimizations

There is a certain set of tasks for which the non-idling optimization can be shown to still be valid. We will call these tasks w-tasks, and all w-tasks τ_i must conform to the following constraints:

Definition 1. *A w-task τ_i must satisfy all the following properties:*

i. *τ_i is not a precursor-task for any dependency in the task-set. This means that q_i-transitions from state s_1 to state s_2 always have the property that $c_{s_1} = c_{s_2}$.*

ii. *τ_i has no non-preemtible section between its first time-slot and its second time-slot. This means that q_i-transitions have no requirement that the smallest number and the second-smallest number in Q have to be consecutive.*

iii. *If τ_i is an event-based dependent task, then it has zero release delay.*

Recall that the definition of the competitive ratio only considers the maximum achievable utility for the offline algorithm on a given sequence of non-dependent and anonymous time-based dependent task releases. This means that any path for which there exists another path with at least the same utility and an identical sequence of non-dependent and anonymous time-based dependent task releases is unnecessary for the computation of the competitive ratio. If for all paths that contain a certain edge in the offline graph, there exists another path with at least equal utility that does not contain this edge and has an identical sequence of non-dependent and anonymous time-based dependent task releases, then this edge can therefore be safely deleted from the offline graph.

This same logic also applies to the transitions that make up each edge: If a certain transition can never under any circumstances be contained in any edge on an optimal path, then it is safe to simply not consider this transition when constructing the offline graph. With this in mind, the non-idling optimization states that:

Theorem 1. *If τ_i is a w-task, then any q_i-transition witch leads to a state s with $p_s(0) = false$ cannot be optimal.*

In order to prove this, some technical details are needed.

Theorem 2. *If two states s_o and s_u have the property that $c_{s_o} = c_{s_u}$, $d_{s_o} = d_{s_u}$, and for all n where $p_{s_u}(n) = false$, $p_{s_o}(n) = false$, then for each path starting in s_u, there must exist a path with an identical sequence of non-dependent and anonymous time-based dependent task releases and identical utility starting in s_o.*

Proof. Each path starting in s_u consists of a sequence of edges. Each edge consists of three possible types of transitions, and for each of these three transition-types, we can construct a transition in an edge starting from s_o in the following way:

- For q_i-transitions from s_u to s'_u, we construct a q_i-transition from s_o to s'_o by using the same set Q as in the original transition. By the assumption, for all n where $p_{s_u}(n) = false$, it must be the case that $p_{s_o}(n) = false$, so this is always possible. Since $c_{s_o} = c_{s_u}$ by the assumption, and since the set Q is identical for both q_i-transitions, it must be the case that $c_{s'_o} = c_{s'_u}$, and that for all n where $p_{s'_u}(n) = false$, it must be the case that $p_{s'_o}(n) = false$.
- For a-transitions from s_u to s'_u, we just insert an a-transition from s_o to s'_o. Since a-transitions just shift all values of p one step closer to zero, the value of p at zero is discarded altogether, and the structure is otherwise left unchanged, it must still remain the case that for all n where $p_{s'_u}(n) = false$, it must be the case that $p_{s'_o}(n) = false$.
- For d_i-transistions from s_u to s'_u, we insert a d_i-transition from s_o to s'_o. This must always be possible, since $d_{s_u} = d_{s_o}$ by the assumption. Since d_i-transition don't interact with the values of p at all, it must still remain the case that for all n where $p_{s'_u}(n) = false$, it must be the case that $p_{s'_o}(n) = false$.

In all three cases, $c_{s'_o} = c_{s'_u}$, $d_{s'_o} = d_{s'_u}$, and for all n where $p_{s'_u}(n) = false$, it must be the case that $p_{s'_o}(n) = false$. Since this property is preserved after each single transition, we can construct an additional transition of the path starting from s_o each time in the same way, which means the property is preserved along the entire path starting from s_o. Also, since the path starting from s_o has the same q_i-transitions in the same order as the path starting from s_u, the two paths must have an identical sequence of non-dependent and anonymous time-based dependent task releases and identical utility. □

Theorem 3. *If there is a path starting from an arbitrary state s that leads to a state s_o, and another path starting from s with the same utility and the same sequence of non-dependent and anonymous time-based dependent task releases that leads to a state s_u, and s_o and s_u have the property that $c_{s_o} = c_{s_u}$, $d_{s_o} = d_{s_u}$, and for all n where $p_{s_u}(n) = false$, $p_{s_o}(n) = false$, then for any execution path of the offline algorithm containing a path from s to s_u, we can construct another execution path of the offline algorithm that contains a path form s to s_o instead, and both paths will have identical* *utility and the same sequence of non-dependent and anonymous time-based dependent task releases.*

Proof. By Theorem 2, for any path that starts in s_u, we can construct a path starting in s_o with the same utility and the same sequence of non-dependent and anonymous time-based dependent task releases, and by assumption the paths from s to s_o and from s to s_u have the same utility and sequence of non-dependent and anonymous time-based dependent task releases. $\square$

Theorem 4. *An edge containting a q_i-transition for a w-task τ_i from an arbitrary and possibly "intermediate" (sub-)state s with $p_s(0) = false$ to a (sub-)state s_u with $p_{s_u}(0) = false$ can never be optimal for w-tasks τ_i.*

Proof. By the definition of q_i-transitions, $p_{s_u}(0) = false$ if and only if $0 \notin Q$ for the q_i-transition from s to s_u. From this transition, we construct a different q_i-transition where $0 \in Q$ to another state s_o by removing the smallest number from the set Q of the original transition, and adding 0 to it. Since τ_i is a w-task, this is always possible: There can be no non-preemptible section between the first and second slots of the task, and there can be no non-zero release delay. Since τ_i is not a precursor task, we also know that $c_s(n) = c_{s_u}(n) = c_{s_o}(n)$ for all n. Since q_i-transitions never modify d_s, we also have $d_{s\tau} = d_{s_u\tau} = d_{s_o\tau}$ for all tasks τ. There are exactly two differences between s_u and s_o: $p_{s_u}(0) = false$ while $p_{s_o}(0) = true$, and $p_{s_u}(q) = true$ while $p_{s_o}(q) = false$, where q is the smallest number in the q_i-transition from s_0 to s_u.

There are two possibilities for the edge of the offline-algorithm containing the q_i-transition from s to s_u: Either there is another q_i-transition with $0 \in Q$ before the next a-transition, or there is not. The first case is impossible, since all q_i-transitions must be ordered descendingly by priority, so a lower priority job would have to be scheduled before a higher priority job, which violates Lemma 1. In the second case, if we apply the same lower priority q_i-transitions and then the a-transition to both s_u and s_o, this will result in states s'_u and s'_o, with $c_{s'_o} = c_{s'_u}$, $d_{s'_o} = d'_{s_u}$, and for all n where $p_{s'_u}(n) = false$, $p_{s'_o}(n) = false$. Therefore, the transition from s to s_u cannot be optimal by Theorem 3. $\square$

CHAPTER 7

Implementation

The valuation part of the framework computes the competitive ratio by finding the minimum mean cycle in the generated graph, as in [CPKS18]. Since this is a computation-heavy problem for larger graphs, the algorithm was parallelized and implemented in CUDA on a GPU to improve performance.

7.1 Madani's algorithm implemented in CUDA

For finding the minimum mean cycle, we use a parallel algorithm based on Madani's algorithm presented in [Mad02] that was developed specifically for this framework. It consists of an initialization phase followed by two rounds of some number of sequential iterations. For a flow chart of this algorithm, see Figure 7.1.

In the initialization phase, the self-loop with minimum weight is used as the upper bound for the weight of the minimum mean cycle. Then all self-loops can be removed from the graph, since it is impossible for a self-loop to be part of a larger minimum mean cycle. The rest of the work is organized in CUDA compute kernels, which access two arrays of node data: One array containing old data (that were written in the previous iteration), and one array containing new data (that is written this iteration, based on the old data). In these arrays, there are two variables stored for each node: The utility (or value) of the node, and the optimal outgoing edge of the node.

At the end of the initialization phase, the old utility is set to zero for all nodes and the old optimal edge is set to the outgoing edge with the smallest index. After that, there are two rounds of value iteration, each consisting of n iterations, where n is the number of nodes in the graph.

In the first round, each iteration starts with setting the utility of all nodes in the new data array to the weight of the previously chosen edge plus the utility of that edge's target node in the old data array. This is done in a kernel with one thread per node.

After that, there is another kernel with one thread per edge, which atomically sets the utility of the source nodes in the new data array to $\min_{e \in out(s)} (u_n(s), u_{o(e)}(t) + w(e))$, where $u_n(s)$ is the previously stored utility of the source node in the new data array, $u_{o(e)}(t)$ is the utility of the target node in the old data array, and $w(e)$ is the weight of the edge. In other words, the utility of each node in the new data array is set to the minimum of all edge weights plus their target node utilities in the old data array. This kernel also sets the optimal edge in the new data array in case the old value was overwritten. After that, the pointers to the new data array and the old data array are switched, so that the new data for this iteration is the old data for the next iteration.

The second round is the same as the first round, except that there is an additional superedge-kernel with one thread per node before the pointers are switched. In this kernel, each node sends a message to all the nodes with optimal edges pointing to it, in other words messages are sent backwards along the optimal edges. Note that in the context of this algorithm, "sending a message" just means that each node saves the origin node, hop-count and collected utility of the message it received in the last iteration, in order to pass it along to the nodes with optimal edges pointing to it in the next iteration, while updating the hop-count and utility in the process. In practice, each node reads the old values from the node its optimal edge points to, and updates its own new values accordingly. This means that messages travel one hop per iteration, and once a node receives a message originally sent by itself, a cycle is found. If it has a smaller mean weight than the previously best cycle, it becomes the new best cycle. In [Mad02], these backwards traveling messages are called superedges.

Each compute kernel has to be entirely completed before the next one can start.

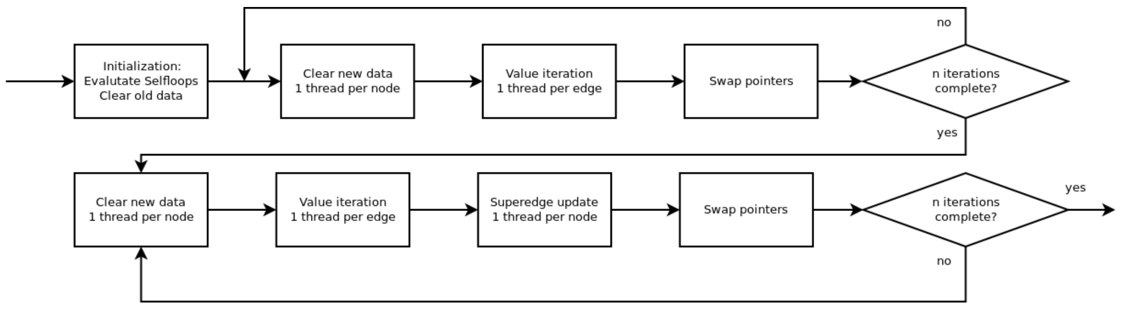

Figure 7.1: Flow chart of the superedge algorithm [Mad02]

CHAPTER 8

Results

In order to demonstrate the utility of our framework, we deliver four different tasksets that represent typical applications. Table 8.1 contains the exact specification of the studied tasksets, and their representation in our framework can be found in the appendix.

8.1 Tasksets

The *packet-switching taskset* models the behavior of a switch, where each task corresponds to a fragment of an incoming packet. The precursor-tasks represent the header part of their respective packets, and since just sending the header and not the body of a packet isn't very useful, has a value of zero. The processing of the header-part is also assumed to be non-preemptible. The taskset contains one low-value task and one high-value task, and both of them are dependent tasks that represent the body of their respective packets. The dependency is time-based, and the unpaired versions of the dependent tasks are not represented to the taskset ($c_i = d_i = v_i = 0$). A taskset like this might be used to analyze an environment like the one studied in [BIS10].

The *sporadic-interrupt taskset* models a normal workload task that occasionally has to respond to an interrupt. The interrupt is represented by a short precursor task, and the workload task is a time-based dependent task that has an unpaired version that represents normal monitoring activity and a paired version that represents the response to an interrupt. Similar problems are presented in [PKB$^+$08] and [TBW94].

The *handshake-protocol taskset* makes the assumption that after a message is sent, there is a reliable response exactly one unit of time later. The task representing the response is therefore modeled as an event-based dependency with a delay of one. This can model situations where communication between two given components is highly reliable, but external messages are not known ahead of time.

taskset	task	C_i	D_i	V_i	properties
packet-switching	τ_1	1	2	0	
	τ_2'	4	6	4	time-based dependency on τ_1
	τ_3	2	3	0	nonpreemptible
	τ_4'	5	8	8	time-based dependency on τ_3
sporadic-interrupt	τ_1	1	2	1	
	τ_2	2	2	1	unpaired, time-based dependency on τ_1
	τ_2'	3	4	6	paired, time-based dependency on τ_1
handshake-protocol	τ_1	2	4	1	
	τ_2	1	3	1	
	τ_3	3	5	10	event-based dependency on τ_1 OR τ_2. Release-delay 1.
query-scheduling	τ_1	2	3	2	
	τ_2	1	1	1	
	τ_3	2	4	0	event-based dependency on τ_1.
	τ_4	1	1	10	event-based dependency on τ_1 AND (τ_2 OR τ_3).

Table 8.1: Studied Tasksets

The *query-scheduling taskset* represents a more complicated monitoring application, where different dependencies between tasks might become relevant. There is one high-value event-based dependent task, which represents successful evaluation of a query. Different combinations of the various precursor tasks, which represent reading new data, processing new requests or evaluating existing data, can be used to trigger the release of this high-value task. This taskset is based on the problem studied in [ZWL13].

8.2 Performance

Table 8.2 shows the results of evaluating each online-algorithm implemented in the framework (see Section 5.1) on each task set, under an unconstrained adversary. These evaluations were run on a computer with INTEL Core i7-5820K, 32GB RAM, EVGA GeForce GTX Titan X SuperClocked 12GB.

Due to the parallelization of the valuation-part of the framework in CUDA, the performance is much better than in the original framework [CPKS18], which means larger graphs like the ones for the packet-switching task set can be analyzed. Note that the only online algorithm with positive competitive ratio for this task set is the precedence-aware profit density algorithm, which achieves $CR = 1$. The profit density algorithm also achieves relatively good results for the other tasksets. This suggests that the competitiveness of online-algorithms under precedence constraints can be significantly improved by making them precendence-aware.

Algorithms that use a latest start time interrupt like Dover and Dstar are particularly harmed by the presence of non-preemptible sections, since the adversary can release jobs

algorithm	task set	nodes	edges	CR	evaluation-time (seconds)
dover	handshake-protocol	9805	65888	$\frac{12}{33}$	3.95
dover	packet-switching	173049	1989068	$\frac{0}{8}$	182.66
dover	query-scheduling	2691	16154	$\frac{16}{65}$	0.87
dover	sporadic-interrupt	179	780	$\frac{10}{22}$	0.05
dstar	handshake-protocol	12878	89117	$\frac{12}{33}$	5.38
dstar	packet-switching	364067	4617035	$\frac{0}{8}$	707.59
dstar	query-scheduling	2677	13984	$\frac{6}{26}$	0.90
dstar	sporadic-interrupt	238	1077	$\frac{1}{7}$	0.07
edf	handshake-protocol	8045	74476	$\frac{2}{11}$	4.05
edf	packet-switching	77697	1388806	$\frac{0}{8}$	50.55
edf	query-scheduling	2126	13683	$\frac{2}{13}$	0.78
edf	sporadic-interrupt	146	768	$\frac{4}{21}$	0.04
fifo	handshake-protocol	6890	50175	$\frac{2}{11}$	3.40
fifo	packet-switching	66265	585085	$\frac{0}{8}$	27.47
fifo	query-scheduling	1692	6095	$\frac{3}{25}$	0.81
fifo	sporadic-interrupt	150	642	$\frac{10}{22}$	0.04
pd	handshake-protocol	5077	39971	$\frac{11}{24}$	1.76
pd	packet-switching	40401	349788	$\frac{1}{1}$	40.62
pd	query-scheduling	1805	9144	$\frac{12}{25}$	0.58
pd	sporadic-interrupt	88	357	$\frac{7}{15}$	0.03
sp	handshake-protocol	4609	49668	$\frac{5}{44}$	1.70
sp	packet-switching	69462	1363534	$\frac{0}{8}$	43.78
sp	query-scheduling	1247	5719	$\frac{2}{13}$	0.42
sp	sporadic-interrupt	123	632	$\frac{4}{22}$	0.05
srt	handshake-protocol	5993	67429	$\frac{2}{11}$	2.83
srt	packet-switching	72284	1309364	$\frac{0}{8}$	46.64
srt	query-scheduling	2095	13734	$\frac{2}{13}$	0.76
srt	sporadic-interrupt	146	768	$\frac{4}{21}$	0.05
sst	handshake-protocol	12216	101343	$\frac{9}{88}$	7.16
sst	packet-switching	125294	2061334	$\frac{0}{8}$	107.76
sst	query-scheduling	1897	9669	$\frac{7}{38}$	0.90
sst	sporadic-interrupt	135	752	$\frac{1}{8}$	0.07

Table 8.2: Results table, including nodes and edges of the product graph, the ratio of the minimum ratio cycle (which equals the CR), and the time in seconds it took to find the minimum ratio cycle.

in such a way that the latest start time interrupt for high value jobs occurs while the algorithm is in a non-preemptible section.

For a graphical representation of these results, see Figure 8.1.

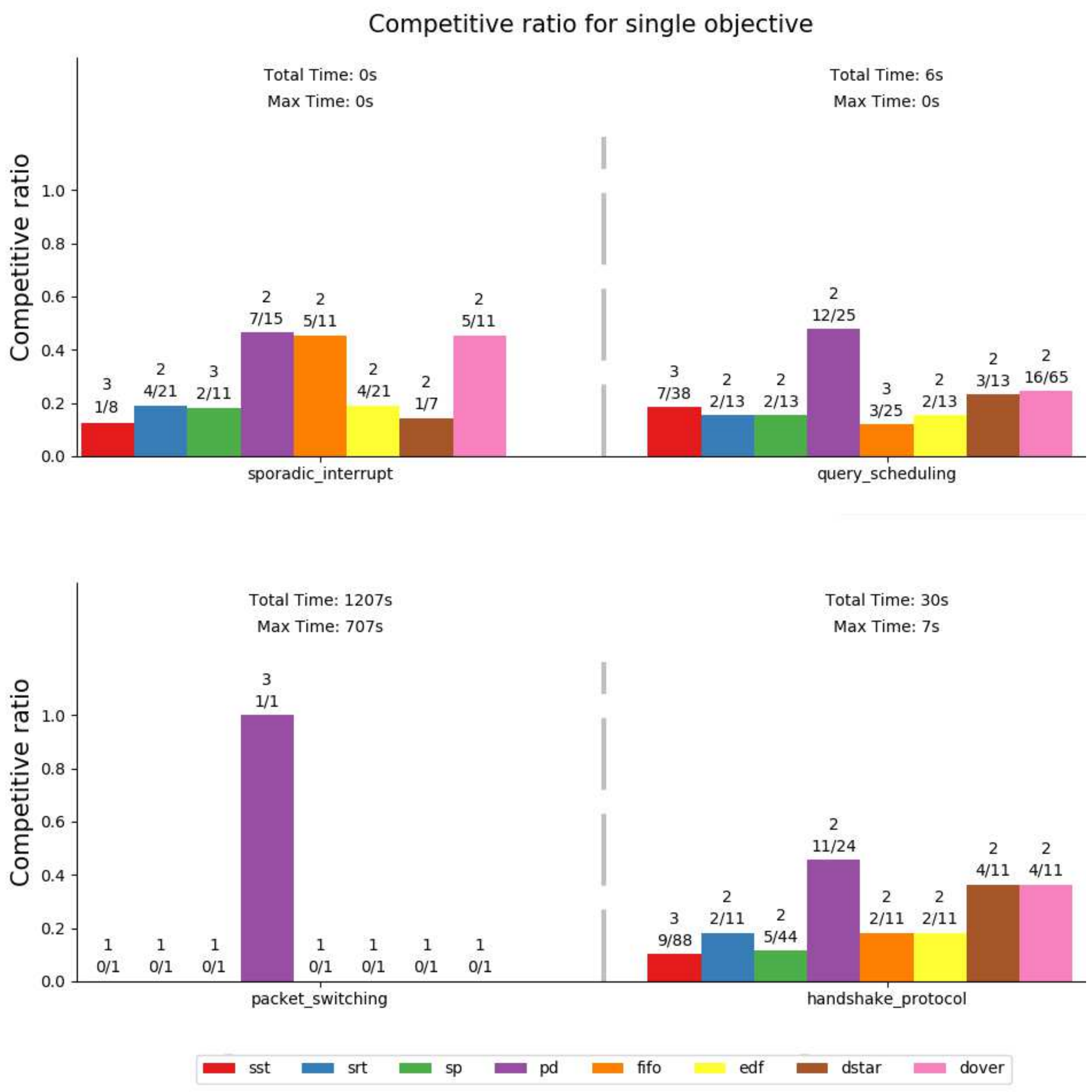

Figure 8.1: Results figure. Each of these four diagrams corresponds to one studied task set, with each bar corresponding to an online algorithm. The fractions above the bars are the competitive ratios that the algorithm achieved on the task set. The numbers above the fractions show how many rounds of guided binary search were necessary to find this competitive ratio.

8.3 Impact of EDF*

Only one of the four studied task sets (query-scheduling) includes a precursor task with a larger relative deadline than its (transitive) dependent tasks. Compared to the offline algorithm based on EDF used in [CPKS18], the algorithm based on EDF* used in this work produces a graph with an equal number of nodes, but about twice as many edges on this task set. However, for all of these additional edges, there exists an identical edge which is also constructed by the EDF-version, which means the redundant edges are discarded during the construction of the product graph. This redundancy is due to the fact that the offline algorithm has to construct an edge for every possible dynamic priority for every precursor task, and the scheduling decision for the released job will often be the same regardless of priority.

CHAPTER 9

Conclusions and Future Work

The approach to automatically computing the competitive ratio of an online algorithm for a given taskset was extended to include non-preemptible sections and precedence constraints. Precedence constraints can be either event-based or time-based, and their dependencies can be expressed as a combination of AND- and OR-joins. Scheduling anomalies arising from this new model were discussed, including how they could impact previous optimizations. Also, the precedence-aware profit-density algorithm was added to the framework and used to evaluate some task sets. With the parallelization of the valuation-part of the framework, performance was increased and larger task sets can be analyzed.

The framework could be further extended by considering additional types of precedences, implementing additional online-algorithms or further performance improvements. The framework still uses Madani's superedge-algorithm presented in [Mad02], but there may be other ways for finding the minimum mean cycle or minimum ratio cycle that are better suited to the GPU. The algorithms presented in [BB15] and [BHK17] may be worth a try for this purpose.

Bibliography

[BB15] Federico Busato and Nicola Bombieri. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. 2015.

[BBMD03] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 253–264, New York, NY, USA, 2003. ACM.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, New York, NY, USA, 1998.

[BHK17] Karl Bringmann, Thomas Dueholm Hansen, and Sebastian Krinninger. Improved algorithms for computing the cycle of minimum cost-to-time ratio in directed graphs. *CoRR*, abs/1704.08122, 2017.

[BIS10] Alan Burns, Leandro Soares Indrusiak, and Zheng Shi. Schedulability analysis for real time on-chip communication with wormhole switching. *Int. J. Embed. Real-Time Commun. Syst.*, 1(2):1–22, April 2010.

[BKM$^+$91] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 100–110, Washington, DC, USA, 1991. IEEE Computer Society.

[BKM$^+$92] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Syst.*, 4(2):125–144, May 1992.

[Bla77] Jacek Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop Organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, Amsterdam, The Netherlands, The Netherlands, 1977. North-Holland Publishing Co.

[BMS10] Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. In *Proceedings of the 18th Annual European Conference on Algorithms: Part II*, ESA'10, pages 230–241, Berlin, Heidelberg, 2010. Springer-Verlag.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer Publishing Company, Incorporated, 3rd edition, 2011.

[CPKS18] Krishnendu Chatterjee, Andreas Pavlogiannis, Alexander Kößler, and Ulrich Schmid. Automated competitive analysis of real-time scheduling with graph games. *Real-Time Syst.*, 54(1):166–207, January 2018.

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Syst.*, 2(3):181–194, September 1990.

[Eve79] Shimon Even. *Graph Algorithms.* W. H. Freeman & Co., New York, NY, USA, 1979.

[Gui] CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[KS95] Gilad Koren and Dennis Shasha. Dover: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. *SIAM J. Comput.*, 24(2):318–339, April 1995.

[LMMW16] Elisabeth Lübbecke, Olaf Maurer, Nicole Megow, and Andreas Wiese. A new approach to online scheduling: Approximating the optimal competitive ratio. *ACM Trans. Algorithms*, 13(1):15:1–15:34, December 2016.

[Mad02] Omid Madani. Polynomial value iteration algorithms for deterministic MDPs. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, pages 311–318, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[PKB$^+$08] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. *SIGOPS Oper. Syst. Rev.*, 42(4):13–25, April 2008.

[SAr$^+$04] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2-3):101–155, November 2004.

[SBHP11] A. Al Sheikh, O. Brun, P. E. Hladik, and B. J. Prabhu. A best-response algorithm for multiprocessor periodic scheduling. In *Proceedings of the*

2011 23rd Euromicro Conference on Real-Time Systems, ECRTS '11, pages 228–237, Washington, DC, USA, 2011. IEEE Computer Society.

[TBW94] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, Mar 1994.

[ZWL13] Yongluan Zhou, Ji Wu, and Ahmed Khan Leghari. Multi-query scheduling for time-critical data stream applications. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 15:1–15:12, New York, NY, USA, 2013. ACM.

Appendix

C code for online algorithms

EDF

```
unsigned int edf_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);
  int scheduled_slots = 0;

  if(scheduled_job != NULL)
  {
    scheduled_slots += scheduled_job[0];
  }

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > 0)
        {
          if(scheduled_slots == 0)
          {
            if(remaining_c > current_d + 1)
            { //This job already has negative laxity, so it es deleted.
              array[current_d] = 0;
            }
            else
            { //schedule this job.
              scheduled_job = &array[current_d];
              scheduled_slots += remaining_c;
            }
          }
          else
          {
            if(&array[current_d] != scheduled_job)
            { //avoid overwriting a nonpreemptible job
              if(remaining_c + scheduled_slots > current_d + 1)
              { //this job will have negative laxity before it is scheduled, so it is deleted.
                array[current_d] = 0;
              }
              else
              { //schedule this job to completion.
                scheduled_slots += remaining_c;
              }
            }
          }
        }
      }
    }
  }
  if(scheduled_job == NULL)
  {
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      update_dependencies(get_generic_dependencies(state), task);
      return(v[task]);
    }
    else
    {
      return(0);
    }
  }
}
```

FIFO

```
unsigned int fifo_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);
  int scheduled_slots = 0;

  if(scheduled_job != NULL)
  {
    scheduled_slots += scheduled_job[0];
  }

  for(int newness = 0; newness < maxd; newness++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      int current_d = newness - maxd + d[task];
      if(current_d >= 0 && current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > 0)
        {
          if(scheduled_slots == 0)
          {
            if(remaining_c > current_d + 1)
            { //This job already has negative laxity, so it es deleted.
              array[current_d] = 0;
            }
            else
            { //schedule this job.
              scheduled_job = &array[current_d];
              scheduled_slots += remaining_c;
            }
          }
          else
          {
            if(&array[current_d] != scheduled_job)
            { //avoid overwriting a nonpreemptible job
              if(remaining_c + scheduled_slots > current_d + 1)
              { //this job will have negative laxity before it is scheduled, so it is deleted.
                array[current_d] = 0;
              }
              else
              { //schedule this job to completion.
                scheduled_slots += remaining_c;
              }
            }
          }
        }
      }
    }
  }

  if(scheduled_job == NULL)
  {
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      update_dependencies(get_generic_dependencies(state), task);
      return(v[task]);
    }
    else
    {
      return(0);
    }
  }
}
```

SP

```
unsigned int sp_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);
  int scheduled_slots = 0;

  if(scheduled_job != NULL)
  {
    scheduled_slots += scheduled_job[0];
  }

  for(int task = 0; task < total_tasknumber(); task++)
  {
    unsigned char* array = get_generic_subarray(state, task);

    for(int current_d = 0; current_d < d[task]; current_d++)
    {
      unsigned char remaining_c = array[current_d];
      if(remaining_c > 0)
      {
        if(scheduled_slots == 0)
        {
          if(remaining_c > current_d + 1)
          { //This job already has negative laxity, so it es deleted.
            array[current_d] = 0;
          }
          else
          { //schedule this job.
            scheduled_job = &array[current_d];
            scheduled_slots += remaining_c;
          }
        }
        else
        {
          if(&array[current_d] != scheduled_job)
          { //avoid overwriting a nonpreemptible job
            if(remaining_c + scheduled_slots > current_d + 1)
            { //this job will have negative laxity before it is scheduled, so it is deleted.
              array[current_d] = 0;
            }
            else
            { //schedule this job to completion.
              scheduled_slots += remaining_c;
            }
          }
        }
      }
    }
  }

  if(scheduled_job == NULL)
  {
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      update_dependencies(get_generic_dependencies(state), task);
      return(v[task]);
    }
    else
    {
      return(0);
    }
  }
}
```

SRT

```
unsigned char* srt_get_minc_job_and_reduce(unsigned char* state, int scheduled_slots)
{
  unsigned char* res = NULL;

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > 0)
        {
          if(remaining_c + scheduled_slots > current_d + 1)
          { //this job will have negative laxity before it is scheduled, so it is deleted.
            array[current_d] = 0;
          }
          else
          {
            if(res == NULL || remaining_c < res[0])
            {
              res = &array[current_d];
            }
          }
        }
      }
    }
  }

  if(res != NULL)
  { //We have the job, now reduce.
    int c_tmp = res[0];
    res[0] = 0;

    srt_get_minc_job_and_reduce(state, scheduled_slots + c_tmp);

    res[0] = c_tmp;
  }

  return(res);
}
```

```
unsigned int srt_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);

  if(scheduled_job != NULL)
  {
    int c_tmp = scheduled_job[0];
    scheduled_job[0] = 0;

    srt_get_minc_job_and_reduce(state, c_tmp);

    scheduled_job[0] = c_tmp;
  }
  else
  {
    scheduled_job = srt_get_minc_job_and_reduce(state, 0);
  }

  if(scheduled_job == NULL)
  {
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      update_dependencies(get_generic_dependencies(state), task);
      return(v[task]);
    }
    else
    {
      return(0);
    }
  }
}
```

PD

```
double pd_get_profit_density(unsigned char* state, unsigned char* job)
{
  int remaining_c = job[0];
  int task = get_generic_tasknumber(state, job);

  double base_value = (double)v[task];
  double base_workload = (double)remaining_c;

  double current_value = base_value;
  double current_workload = base_workload;
  double current_pd = base_value/base_workload;

  uintmax_t dep_bit = 1;
  dep_bit = dep_bit << task;
  uintmax_t* dependency_state = get_generic_dependencies(state);

  for(int dep_task = 0; dep_task < total_tasknumber(); dep_task++)
  {
    if(get_dependency_category(dep_task) == EVENTBASED ||
    get_dependency_category(dep_task) == TIMEBASED_PAIRED)
    {
      int dep_index = get_dependency_index(dep_task);
      if((dep[dep_index][0] & dep_bit) != 0 && (dependency_state[dep_index] & dep_bit) == 0)
      {
        if((base_value + (double)v[dep_task]) / (base_workload + (double)c[dep_task]) > current_pd)
        {
          current_value = base_value + (double)v[dep_task];
          current_workload = base_workload + (double)c[dep_task];
          current_pd = current_value / current_workload;
        }
      }
    }
  }

  return(current_pd);
}


unsigned char* pd_get_maxpd_job_and_reduce(unsigned char* state, int scheduled_slots)
{
  unsigned char* res = NULL;

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > 0)
        {
          if(remaining_c + scheduled_slots > current_d + 1)
          { //this job will have negative laxity before it is scheduled, so it is deleted.
            array[current_d] = 0;
          }
          else
          {
            if(res == NULL || pd_get_profit_density(state, &array[current_d]) >
            pd_get_profit_density(state, res))
            {
              res = &array[current_d];
            }
          }
        }
      }
    }
  }

  if(res != NULL)
  { //We have the job, now reduce.
    int c_tmp = res[0];
    res[0] = 0;

    pd_get_maxpd_job_and_reduce(state, scheduled_slots + c_tmp);

    res[0] = c_tmp;
  }

  return(res);
}


unsigned int pd_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);

  if(scheduled_job != NULL)
  {
    int c_tmp = scheduled_job[0];
    scheduled_job[0] = 0;

    pd_get_maxpd_job_and_reduce(state, c_tmp);

    scheduled_job[0] = c_tmp;
  }
  else
  {
    scheduled_job = pd_get_maxpd_job_and_reduce(state, 0);
  }

  if(scheduled_job == NULL)
  {
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      update_dependencies(get_generic_dependencies(state), task);
      return(v[task]);
    }
    else
    {
      return(0);
    }
  }
}
```

SST

```
unsigned char* sst_get_smallest_slack_time_job_and_discard_negative_laxity(unsigned char* state)
{
  unsigned char* res = NULL;
  int current_smallest_slack_time = INT32_MAX;

  for(int task = 0; task < total_tasknumber(); task++)
  {
    unsigned char* array = get_generic_subarray(state, task);

    for(int current_d = 0; current_d < d[task]; current_d++)
    {
      unsigned char remaining_c = array[current_d];
      if(remaining_c > 0)
      {
        if(remaining_c > current_d + 1)
        { //This job already has negative laxity, so it es deleted.
          array[current_d] = 0;
        }
        else
        { //Compare with previous best
          if(current_d + 1 - remaining_c < current_smallest_slack_time)
          {
            current_smallest_slack_time = current_d + 1 - remaining_c;
            res = &array[current_d];
          }
        }
      }
    }
  }

  return(res);
}


bool sst_check_job_is_relevant(unsigned char* state, int task, int current_d)
{
  if(current_d == 0)
  {
    return(false);
  }
  else
  {
    int frontiermultiplier = (1 << (numTasks_independent + numTasks_timebased));

    for(unsigned int i = 0; i < frontiermultiplier; i++)
    {
      unsigned char* newstate = advance_state(state, i);

      unsigned char* scheduled_job = get_generic_nonpreemptible_job(newstate);
      if(scheduled_job == NULL)
      {
        scheduled_job = sst_get_smallest_slack_time_job_and_discard_negative_laxity(newstate);
      }
      if(scheduled_job != NULL)
      {
        scheduled_job[0]--;
        if(scheduled_job[0] == 0)
        {
          int task = get_generic_tasknumber(state, scheduled_job);
          update_dependencies(get_generic_dependencies(state), task);
        }
      }

      unsigned char* newarray = get_generic_subarray(newstate, task);

      if(&newarray[current_d-1] == scheduled_job)
      {
        free(newstate);
        return(true);
      }

      if(newarray[current_d-1] != 0 && sst_check_job_is_relevant(newstate, task, current_d-1))
      {
        free(newstate);
        return(true);
      }

      free(newstate);
    }

    return(false);
  }
}


void sst_reduce(unsigned char* state)
{
  unsigned char original_state[statesize];
  memcpy(original_state, state, statesize);

  for(int task = 0; task < total_tasknumber(); task++)
  {
    unsigned char* array = get_generic_subarray(state, task);

    for(int current_d = 0; current_d < d[task]; current_d++)
    {
      unsigned char remaining_c = array[current_d];
      if(remaining_c > 0 && !sst_check_job_is_relevant(original_state, task, current_d))
      {
        array[current_d] = 0;
      }
    }
  }
}


unsigned int sst_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);

  if(scheduled_job == NULL)
  {
    scheduled_job = sst_get_smallest_slack_time_job_and_discard_negative_laxity(state);
  }

  if(scheduled_job == NULL)
  {
    sst_reduce(state);
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      update_dependencies(get_generic_dependencies(state), task);
      sst_reduce(state);
      return(v[task]);
    }
    else
    {
      sst_reduce(state);
      return(0);
    }
  }
}
```

DOVER

```
unsigned char* dover_get_Qrecent_array(unsigned char* state)
{
  unsigned char* subarrays = &state[total_dependencynumber()*sizeof(uintmax_t)];

  for(int task = 0; task < total_tasknumber(); task++)
  {
    subarrays = &subarrays[generic_subarray_size(task)];
  }

  return(subarrays);
}


int dover_get_current_availtime(unsigned char* state)
{
  int res = INT32_MAX;

  unsigned char* qrecent_array = dover_get_Qrecent_array(state);
  for(int task = 0; task < total_tasknumber(); task++)
  {
    if(qrecent_array[task] != 255)
    {
      unsigned char* array = get_generic_subarray(state, task);

      res = res - array[qrecent_array[task]];
      int cmp = qrecent_array[task] - array[qrecent_array[task]] + 1;
      res = (res > cmp) ? cmp : res;
    }
  }

  return(res);
}


unsigned char* dover_get_first_qother_task(unsigned char* state)
{
  unsigned char* qrecent_array = dover_get_Qrecent_array(state);

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > 0 && qrecent_array[task] != current_d)
        {
          return(&array[current_d]);
        }
      }
    }
  }

  return(NULL);
}

unsigned char* dover_get_first_qrecent_task(unsigned char* state)
{
  unsigned char* qrecent_array = dover_get_Qrecent_array(state);

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > 0 && qrecent_array[task] == current_d)
        {
          return(&array[current_d]);
        }
      }
    }
  }

  return(NULL);
}


void dover_decide_between_qrecent_and_qother(unsigned char* state, unsigned char* qother_task)
{
  if(qother_task != NULL)
  {
    unsigned char* first_qrecent_task = dover_get_first_qrecent_task(state);
    int availtime = dover_get_current_availtime(state);

    if(first_qrecent_task == NULL ||
    (get_generic_deadline(state, qother_task) <
    get_generic_deadline(state, first_qrecent_task) && qother_task[0] <= availtime))
    {
      unsigned char* qrecent_array = dover_get_Qrecent_array(state);
      qrecent_array[get_generic_tasknumber(state, qother_task)] = get_generic_deadline(state, qother_ta
    }
  }
}


void dover_task_completion_interrupt(unsigned char* state)
{
  unsigned char* first_qother_task = dover_get_first_qother_task(state);
  dover_decide_between_qrecent_and_qother(state, first_qother_task);
}


unsigned char* dover_advance_state(unsigned char* oldstate, unsigned int workload)
{
  unsigned char* res = generic_advance_state(oldstate, workload);
  unsigned char* Qrecent_old = dover_get_Qrecent_array(oldstate);
  unsigned char* Qrecent_new = dover_get_Qrecent_array(res);

  for(int i = 0; i < total_tasknumber(); i++)
  {
    Qrecent_new[i] = (Qrecent_old[i] == 0 || Qrecent_old[i] == 255) ? 255 : Qrecent_old[i] - 1;
  }

  return(res);
}
```

```
unsigned int dover_get_qrecent_value_sum(unsigned char* state)
{
  unsigned int res = 0;

  unsigned char* qrecent_array = dover_get_Qrecent_array(state);
  for(int task = 0; task < total_tasknumber(); task++)
  {
    if(qrecent_array[task] != 255)
    {
      res += v[task];
    }
  }

  return(res);
}


void dover_latest_start_time_interrupt(unsigned char* state)
{
  unsigned char* qrecent_array = dover_get_Qrecent_array(state);
  double magicnumber = 2.0;

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c == current_d+1 && qrecent_array[task] != current_d)
        {
          if((double)v[task] > magicnumber * (double)dover_get_qrecent_value_sum(state))
          {
            for(int i = 0; i < total_tasknumber(); i++)
            {
              qrecent_array[i] = 255;
            }
            qrecent_array[task] = current_d;
          }
          else
          {
            array[current_d] = 0;
          }
        }
      }
    }
  }
}


void dover_reduce_simple(unsigned char* state)
{
  unsigned char* qrecent_array = dover_get_Qrecent_array(state);

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > current_d && qrecent_array[task] != current_d)
        {
          array[current_d] = 0;
        }
      }
    }
  }
}


bool dover_check_job_is_relevant(unsigned char* state, int task, int current_d)
{
  if(current_d == 0)
  {
    return(false);
  }
  else
  {
    int frontiermultiplier = (1 << (numTasks_independent + numTasks_timebased));

    for(unsigned int i = 0; i < frontiermultiplier; i++)
    {
      unsigned char* newstate = advance_state(state, i);

      unsigned char* scheduled_job = get_generic_nonpreemptible_job(newstate);
      if(scheduled_job == NULL)
      {
        for(int task = 0; task < total_tasknumber(); task++)
        {
          unsigned char* array = get_generic_subarray(newstate, task);
          if(d[task] > 0 && array[d[task]-1] != 0)
          {
            dover_decide_between_qrecent_and_qother(newstate, &array[d[task]-1]);
          }
        }

        dover_latest_start_time_interrupt(newstate);

        scheduled_job = dover_get_first_qrecent_task(newstate);
      }
      if(scheduled_job != NULL)
      {
        scheduled_job[0]--;
        if(scheduled_job[0] == 0)
        {
          int task = get_generic_tasknumber(newstate, scheduled_job);
          unsigned char* qrecent_array = dover_get_Qrecent_array(newstate);
          qrecent_array[task] = 255;

          update_dependencies(get_generic_dependencies(newstate), task);
          dover_task_completion_interrupt(newstate);
        }
        dover_reduce_simple(newstate);
      }

      unsigned char* newarray = get_generic_subarray(newstate, task);

      if(&newarray[current_d-1] == scheduled_job)
      {
        free(newstate);
        return(true);
      }

      if(newarray[current_d-1] != 0 && dover_check_job_is_relevant(newstate, task, current_d-1))
      {
        free(newstate);
        return(true);
      }

      free(newstate);
    }

    return(false);
  }
}
```

```
void dover_reduce(unsigned char* state)
{
  unsigned char original_state[statesize];
  memcpy(original_state, state, statesize);

  unsigned char* qrecent_array = dover_get_Qrecent_array(state);

  for(int task = 0; task < total_tasknumber(); task++)
  {
    unsigned char* array = get_generic_subarray(state, task);

    for(int current_d = 0; current_d < d[task]; current_d++)
    {
      unsigned char remaining_c = array[current_d];
      if(remaining_c > 0 && qrecent_array[task] != current_d)
      {
        if(remaining_c > current_d || !dover_check_job_is_relevant(original_state, task, current_d))
        {
          array[current_d] = 0;
        }
      }
    }
  }
}


unsigned int dover_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);

  if(scheduled_job == NULL)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      unsigned char* array = get_generic_subarray(state, task);
      if(d[task] > 0 && array[d[task]-1] != 0)
      {
        dover_decide_between_qrecent_and_qother(state, &array[d[task]-1]);
      }
    }

    dover_latest_start_time_interrupt(state);

    scheduled_job = dover_get_first_qrecent_task(state);
  }

  if(scheduled_job == NULL)
  {
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      unsigned char* qrecent_array = dover_get_Qrecent_array(state);
      qrecent_array[task] = 255;

      update_dependencies(get_generic_dependencies(state), task);
      dover_task_completion_interrupt(state);
      dover_reduce(state);
      return(v[task]);
    }
    else
    {
      dover_reduce(state);
      return(0);
    }
  }
}


unsigned char* dover_initial_state(void)
{
  statesize = total_dependencynumber()*sizeof(uintmax_t);

  for(int task = 0; task < total_tasknumber(); task++)
  {
    statesize += generic_subarray_size(task);
  }

  statesize += total_tasknumber(); //Qrecent-indices

  unsigned char* res = (unsigned char*)malloc(statesize);
  memset(res, 0, statesize);

  unsigned char* Qrecent = dover_get_Qrecent_array(res);
  for(int i = 0; i < total_tasknumber(); i++)
  {
    Qrecent[i] = 255;
  }

  return(res);
}
```

DSTAR

```
unsigned char* dstar_get_current_task(unsigned char* state)
{
  int* res = (int*)dover_get_Qrecent_array(state); //This function also works here.
  int task = res[0];
  int current_d = res[1];
  if(task != -1 && current_d != -1)
  {
    unsigned char* array = get_generic_subarray(state, task);
    return(&array[current_d]);
  }
  else
  {
    return(NULL);
  }
}


void dstar_set_current_task(unsigned char* state, unsigned char* job)
{
  int* res = (int*)dover_get_Qrecent_array(state); //This function also works here.
  if(job != NULL)
  {
    res[0] = get_generic_tasknumber(state, job);
    res[1] = get_generic_deadline(state, job);
  }
  else
  {
    res[0] = -1;
    res[1] = -1;
  }
}


int dstar_get_preempt_value(unsigned char* state)
{
  int* res = (int*)dover_get_Qrecent_array(state); //This function also works here.
  return(res[2]);
}
```

```
void dstar_set_preempt_value(unsigned char* state, int preempt_value)
{
  int* res = (int*)dover_get_Qrecent_array(state); //This function also works here.
  res[2] = preempt_value;
}


unsigned char* dstar_advance_state(unsigned char* oldstate, unsigned int workload)
{
  unsigned char* res = generic_advance_state(oldstate, workload);

  dstar_set_preempt_value(res, dstar_get_preempt_value(oldstate));

  unsigned char* old_current_task = dstar_get_current_task(oldstate);
  if(old_current_task == NULL)
  {
    dstar_set_current_task(res, NULL);
  }
  else
  {
    int oldtask = get_generic_tasknumber(oldstate, old_current_task);
    int old_d = get_generic_deadline(oldstate, old_current_task);

    if(old_d == 0)
    {
      dstar_set_current_task(res, NULL);
    }
    else
    {
      unsigned char* newarray = get_generic_subarray(res, oldtask);
      unsigned char* newjob = &newarray[old_d-1];

      dstar_set_current_task(res, newjob);
    }
  }

  return(res);
}


unsigned char* dstar_initial_state(void)
{
  statesize = total_dependencynumber()*sizeof(uintmax_t);

  for(int task = 0; task < total_tasknumber(); task++)
  {
    statesize += generic_subarray_size(task);
  }

  statesize += 3*sizeof(int); //special state.

  unsigned char* res = (unsigned char*)malloc(statesize);
  memset(res, 0, statesize);

  dstar_set_current_task(res, NULL);
  dstar_set_preempt_value(res, 0);

  return(res);
}


void dstar_insert_new_job(unsigned char* state, int task)
{
  unsigned char* current_task = dstar_get_current_task(state);
  unsigned char* array = get_generic_subarray(state, task);

  if(current_task == NULL ||
  (get_generic_deadline(state, current_task) >= d[task]-1 && dstar_get_preempt_value(state) == 0))
  {
    dstar_set_current_task(state, &array[d[task]-1]);
  }
}


void dstar_latest_start_time_interrupt(unsigned char* state)
{
  for(int task = 0; task < total_tasknumber(); task++)
  {
    unsigned char* array = get_generic_subarray(state, task);

    for(int current_d = 0; current_d < d[task]; current_d++)
    {
      unsigned char remaining_c = array[current_d];
      unsigned char* current_task = dstar_get_current_task(state);
      unsigned char* new_job = &array[current_d];
      if(remaining_c == current_d+1 && new_job != current_task)
      {
        if(current_task == NULL || current_task[0] < get_generic_deadline(state, current_task)+1)
        { //current task has slack time.
          dstar_set_current_task(state, new_job);
        }
        else
        {
          int current_task_value = v[get_generic_tasknumber(state, current_task)];
          if(v[task] > dstar_get_preempt_value(state) + current_task_value)
          { //preempt and drop current task, execute lst task.
            current_task[0] = 0;
            dstar_set_preempt_value(state, dstar_get_preempt_value(state) + current_task_value);
            dstar_set_current_task(state, new_job);

          }
          else
          { //Drop lst task.
            new_job[0] = 0;
          }
        }
      }
    }
  }
}


void dstar_task_completion_interrupt(unsigned char* state)
{
  dstar_set_preempt_value(state, 0);

  for(int current_d = 0; current_d < maxd; current_d++)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      if(current_d < d[task])
      {
        unsigned char* array = get_generic_subarray(state, task);
        unsigned char remaining_c = array[current_d];
        if(remaining_c > 0)
        {
          dstar_set_current_task(state, &array[current_d]);
          return;
        }
      }
    }
  }

  dstar_set_current_task(state, NULL);
}


void dstar_reduce_simple(unsigned char* state)
{
  for(int current_d = 0; current_d < maxd; current_d++)
```

```
    {
      for(int task = 0; task < total_tasknumber(); task++)
      {
        if(current_d < d[task])
        {
          unsigned char* array = get_generic_subarray(state, task);
          unsigned char remaining_c = array[current_d];
          if(remaining_c > current_d && dstar_get_current_task(state) != &array[current_d])
          {
            array[current_d] = 0;
          }
        }
      }
    }
}


bool dstar_check_job_is_relevant(unsigned char* state, int task, int current_d)
{
  if(current_d == 0)
  {
    return(false);
  }
  else
  {
    int frontiermultiplier = (1 << (numTasks_independent + numTasks_timebased));

    for(unsigned int i = 0; i < frontiermultiplier; i++)
    {
      unsigned char* newstate = advance_state(state, i);

      unsigned char* scheduled_job = get_generic_nonpreemptible_job(newstate);
      if(scheduled_job == NULL)
      {
        for(int task = 0; task < total_tasknumber(); task++)
        {
          unsigned char* array = get_generic_subarray(newstate, task);
          if(d[task] > 0 && array[d[task]-1] != 0)
          {
            dstar_insert_new_job(newstate, task);
          }
        }

        dstar_latest_start_time_interrupt(newstate);

        scheduled_job = dstar_get_current_task(newstate);
      }
      if(scheduled_job != NULL)
      {
        int task = get_generic_tasknumber(newstate, scheduled_job);
        scheduled_job[0]--;

        if(scheduled_job[0] == 0)
        {
          dstar_task_completion_interrupt(newstate);
          update_dependencies(get_generic_dependencies(newstate), task);
        }
        dstar_reduce_simple(newstate);
      }

      unsigned char* newarray = get_generic_subarray(newstate, task);

      if(&newarray[current_d-1] == scheduled_job)
      {
        free(newstate);
        return(true);
      }

      if(newarray[current_d-1] != 0 && dstar_check_job_is_relevant(newstate, task, current_d-1))
      {
        free(newstate);
        return(true);
      }

      free(newstate);
    }

    return(false);
  }
}


void dstar_reduce(unsigned char* state)
{
  unsigned char original_state[statesize];
  memcpy(original_state, state, statesize);

  for(int task = 0; task < total_tasknumber(); task++)
  {
    unsigned char* array = get_generic_subarray(state, task);

    for(int current_d = 0; current_d < d[task]; current_d++)
    {
      unsigned char remaining_c = array[current_d];
      if(remaining_c > 0 && dstar_get_current_task(state) != &array[current_d])
      {
        if(remaining_c > current_d || !dstar_check_job_is_relevant(original_state, task, current_d))
        {
          array[current_d] = 0;
        }
      }
    }
  }
}


unsigned int dstar_schedule_and_reduce(unsigned char* state)
{
  unsigned char* scheduled_job = get_generic_nonpreemptible_job(state);

  if(scheduled_job == NULL)
  {
    for(int task = 0; task < total_tasknumber(); task++)
    {
      unsigned char* array = get_generic_subarray(state, task);
      if(d[task] > 0 && array[d[task]-1] != 0)
      {
        dstar_insert_new_job(state, task);
      }
    }

    dstar_latest_start_time_interrupt(state);

    scheduled_job = dstar_get_current_task(state);
  }

  if(scheduled_job == NULL)
  {
    return(0);
  }
  else
  {
    int task = get_generic_tasknumber(state, scheduled_job);
    scheduled_job[0]--;

    if(scheduled_job[0] == 0)
    {
      dstar_task_completion_interrupt(state);
      update_dependencies(get_generic_dependencies(state), task);
      dstar_reduce(state);
      return(v[task]);
    }
    else
```

```
      {
        dstar_reduce(state);
        return(0);
      }
    }
}
```

Studied tasksets

```
{
        "packet_switching":
        [
                {"id":"0", "c":1, "d":2, "v":0},
                {"id":"1", "c":4, "d":6, "v":4, "time_dep":"0"},
                {"id":"2", "c":2, "d":3, "v":0, "np":"0"},
                {"id":"3", "c":5, "d":8, "v":8, "time_dep":"2"}
        ],

        "sporadic_interrupt":
        [
                {"id":"0", "c":1, "d":2, "v":1},
                {"id":"1", "c":3, "d":4, "v":6, "time_dep":"0", "c2":2, "d2":2, "v2":1}
        ],

        "handshake_protocol":
        [
                {"id":"0", "c":2, "d":4, "v":1},
                {"id":"1", "c":1, "d":3, "v":1},
                {"id":"2", "c":3, "d":5, "v":10, "event_dep":"0|1", "event_delay":"1"}
        ],

        "query_scheduling":
        [
                {"id":"0", "c":2, "d":3, "v":2},
                {"id":"1", "c":1, "d":1, "v":1},
                {"id":"2", "c":2, "d":4, "v":0, "event_dep":"0"},
                {"id":"3", "c":1, "d":1, "v":10, "event_dep":"0&(1|2)"}
        ]
}
```

How to use the framework

After all the binaries have been built, the framework can be run with the following commands:

Graph generation:

```
$ python main.py -d [graphs-directory] -s [schedulers] -t [taskset]
```

Valuation:

```
$ python main_single.py [graphs-directory] result.out
```

Example:

```
$ python main.py -d ../graphs -s edf,fifo,sp,srt,pd -t ../taskset.json
$ python main_single.py ../graphs result.out
```

The file result.out will then contain the results of the evaluation, including the competitive ratio and the cycle in the evaluated graph.

The schedulers for the graph generation are edf, fifo, sp, srt, pd, sst, dover, dstar. Multiple schedulers can be specified simultaneously, separated by a comma, but without spaces. Note that the content of the specified graphs-directory will be overwritten by the graph generation call.