Masters Thesis

Scheduling High Criticality Real-Time Systems

Humberto Carvalho

CISTER-TR-171102

2017/11/17

Scheduling High Criticality Real-Time Systems

Humberto Carvalho

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: hjesc@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Cyclic executives are used to schedule safety-critical real-time systems because of their determinism, simplicity, and efficiency. One major challenge of the cyclic executive model is to produce the cyclic scheduling timetable. This problem is related to the bin-packing problem and is NP-Hard in the strong sense. Unnecessary context switches within the scheduling table can introduce significant overhead; in IMA (Integrated Modular Avionics), cache-related overheads can increase task execution times up to 33%.

Developed in the context of the Software Engineering Master 19s Degree at ISEP, the Polytechnic Institute of Engineering in Porto Portugal, this thesis contains two contributions to the scheduling literature. The first is a precise and exact approach to computing the slack of a job set that is schedule policy independent. The method introduces several operations to update and maintain the slack at runtime, ensuring the slack of all jobs is valid and coherent. The second contribution is the definition of a state-of-the-art preemptive scheduling algorithm focused on minimizing the number of system preemptions for real-time safety-critical applications within a reasonable amount of time.

Both contributions have been implemented and extensively tested in scala. Experimental results suggest our scheduling algorithm has similar non-preemptive schedulability ratio than Chain Window RM, yet lower ratio in high utilizations than Chain Window EDF and BB-Moore. For task sets that failed to be scheduled non-preemptively, 98-99% of all jobs are scheduled without preemptions. Considering the fact that our scheduler is preemptive, being able to compete with non-preemptive schedulers is an excellent result indeed. In terms of execution time, our proposal is an order of magnitude faster than the aforementioned algorithms. Both contributions of this work are planned to be presented at future conferences such as RTSS@Work and RTAS.

Polytechnic Institute of Engineering
Porto, Portugal
research center in real-time & embedded computing systems

Msc. Thesis

Scheduling High Criticality Real-Time Systems

Humberto Carvalho
1120409

Supervised by

Geoffrey Nelissen Eduardo Tovar

December 17, 2017

To my family, friends, and girls with messy hair and thirsty hearts, that inspired me and without whom none of my success would be possible, all of whom will never read this.

Acknowledgments

The work presented in this report is the result of several months of research, however as Isaac Newton once said “If I have seen further than others, it is by standing upon the shoulders of giants”, this work would not have been possible without the intervention of several key individuals and institutions, who deserve my greatest gratitude.

I would like to begin by expressing my deepest appreciation to my advisor Geoffrey Nelissen, whose ingenuity, knowledge and wisdom forged this project to what it is today. Geoffrey continually and persuasively conveyed a spirit of adventure in regard to the development and success of this project, to say his guidance made a significant impact on the project would be the understatement of the year. Additionally, the wicked chocolates he brought from Brussels were just phenomenal.

I am immensely grateful to Eduardo Tovar, for believing in me and offering the possibility of developing this *far-reaching* project, along with the scholarship granted to me as a reward for that work. It is an honor to have my skills recognized by, and be advised by one of the leading individuals behind real time research.

To my research colleagues at CISTER, whose hospitality and support provided insight and expertise that greatly assisted this project.

To ISEP, the Polytechnic Institute of Engineering in Porto, Portugal, and all of its members, whose dedication and professionalism, not only made ISEP a reference Institute in Portugal, but formerly recognized by several European quality assurance agencies such as EUR-ACE. If I have somehow gone further than others, it is a fallout of their hard work.

Douglas Merril once said ”All of us are smarter than any of us“, I have always lived by that philosophy, and would like to show my appreciation to my closest friends and colleagues whose presence had a significant impact on my personal and academic life, and took my farther than I could have hoped. In highlight my best friend Raquel Cangueiro – whom according to her, deserves heaven for putting up with me these last 10 years –, Fernando Nuno, José Fernandes, Jorge Correia, Nuno Melo, Pedro Silva,and José Vasco, to whom I can only hope I have been as good of a friend as they have been to me.

Finally but not least I would like to thank my family who have always believed in me, including but not limited to, my dad Humberto Carvalho, my mother Cristina Carvalho, my grand mother Maria de Lourdes, my grand father Humberto Carvalho and finally my grand mother Maria Carvalho, for all the unconditional support and for always encouraging me to follow my heart.

Abstract

Cyclic executives are used to schedule safety-critical real-time systems because of their determinism, simplicity, and efficiency. One major challenge of the cyclic executive model is to produce the cyclic scheduling timetable. This problem is related to the bin-packing problem [34] and is *NP-Hard* in the strong sense. Unnecessary context switches within the scheduling table can introduce significant overhead; in IMA (Integrated Modular Avionics), cache-related overheads can increase task execution times up to 33% [18].

Developed in the context of the Software Engineering Master's Degree at ISEP, the Polytechnic Institute of Engineering in Porto Portugal, this thesis contains two contributions to the scheduling literature. The first is a precise and exact approach to computing the slack of a job set that is schedule policy independent. The method introduces several operations to update and maintain the slack at runtime, ensuring the slack of all jobs is valid and coherent. The second contribution is the definition of a state-of-the-art preemptive scheduling algorithm focused on minimizing the number of system preemptions for real-time safety-critical applications within a reasonable amount of time.

Both contributions have been implemented and extensively tested in scala. Experimental results suggest our scheduling algorithm has similar *non-preemptive* schedulability ratio than Chain Window RM [69], yet lower ratio in high utilizations than Chain Window EDF [69] and BB-Moore [68]. For task sets that failed to be scheduled non-preemptively, 98-99% of all jobs are scheduled without preemptions. Considering the fact that our scheduler is preemptive, being able to compete with non-preemptive schedulers is an excellent result indeed. In terms of execution time, our proposal is multiple orders of magnitude faster than the aforementioned algorithms. Both contributions of this work are planned to be presented at future conferences such as RTSS@Work and RTAS.

Keywords: real-time, safety-critical systems, scheduling, ARINC-653, minimizing preemptions, slack computation.

Contents

Chapter 1

Introduction

Many computing systems are responsible for real-time processing where failure can result in catastrophic damages or even the loss of human lives. The safety requirements placed upon such systems are extremely strict, and the deployed solutions tend to be simple so as to facilitate the certification process through which they must pass [42].

Avionics is a typical example of safety critical systems. In order to reduce maintenance and development costs and to improve reusability and evolutivity, aircraft embedded functions (e.g., flight control, flight management, navigation, or systems control) are developed according to the Integrated Modular Avionics (IMA) concept [5].

In IMA different functions are developed independently and later integrated into the same computing nodes. Software functions are assigned to partitions that virtualize their computing environment. Partitions guarantee time and space isolation between functions, preventing faults from propagating from one application to another. The processing resources of the computing platform are distributed between partitions according to a fixed schedule: a cyclic timetable where the execution time intervals of all partitions are defined by the system integrator before deployment.

Yet, building those partition scheduling tables is a NP-hard problem [34] for which several parameters must be taken into consideration: (1) each partition must receive enough processing time to ensure that all its tasks are completed within their timing constraints; (2) the size of the table must be kept sufficiently small to not waste valuable memory space; (3) precedence and mutual exclusion constraints between partitions must be respected; (4) context switches among partitions are costly and should be minimized as to avoid wasting processing resources.

Currently, the partition scheduling tables are mainly built based on the experience acquired by experts. The ultimate goal of this thesis is to automate this process by developing algorithms that generate the partition scheduling tables.

The problem is related to the general scheduling problem and has been significantly addressed by the research community. Several works proposed in the literature range from exact algorithms that achieve optimality by iterating the solution space like branch and bound and mixed-integer programming (MILP) formulations, to meta-heuristics which produce sub-optimal results via optimization algorithms such as simulated annealing and ant colony optimization. All of them approach the scheduling problem in a unique way, and yet, all of them share the same drawback: they are exponential in relation to time and space. The computational demand to produce a solution increases exponentially regarding the number of tasks and constraints in the system, such that the response times of medium-sized task sets are measured in hours, and larger sets become computationally infeasible.

As far as we are aware, none of the solutions in the state-of-the-art completely address the problem of producing an optimal partition schedule where preemptions are minimized within an acceptable time frame. All submissions define optimality as any solution that effectively schedules the entirety of the task set. The authors of this thesis extend this definition of optimality by adding one more constraint: a schedule is deemed optimal if for a given task set, the schedule is feasible and the number of context switches is minimized. By extracting system properties that guide exact or meta-heuristic algorithms towards better solutions, the solution space may be sufficiently pruned to make the problem treatable within an acceptable time frame. Through this unique and novel approach where heuristics meet search space algorithms, the authors of this thesis intend on developing a solution which addresses this scheduling problem.

The contribution of this work is twofold. First, by maximizing processor utilization within each computing node, fewer nodes are required to schedule a system. In a competitive industry where a 1% efficiency increase can translate into hundreds of thousands in yearly fuel savings [106], this work presents a significant contribution. Through those fuel savings, global Carbon Dioxide (CO_2) emissions can be reduced, contributing to the well-being of our home planet and helping comply with the always more stringent regulations defined by the governing authorities. In addition, the results of this work are also a significant contribution to scheduling theory. We intend on extracting scheduling properties that can be charted onto optimal algorithms or heuristics which can guide other solutions efficiently, effectively spanning our contribution to other sub-areas within the scheduling literature.

1.1 Document Structure

The report is divided into four chapters. The first chapter, *Introduction* gives a brief summary of the entire work. The second chapter, named *Outlook* discusses the work context and introduces high criticality real-time systems, emphasizing the avionics sector. The chapter ends with a formal presentation of the problem. *State of the Art*, the third chapter of this report, covers a formal review of state-of-the-art schedulers. It begins with an introduction to the cyclic executive model using online scheduling as an analogy and ends with a review of related work in literature. The fourth chapter, *Project Plan*, presents the architecture and reference implementation of a new state-of-the-art scheduling algorithm. Chapter five introduces two state-of-the-art algorithms, one two compute the slack of a task set, and another to schedule that task set while minimizing the number of system preemptions. Chapter six describes the developed implementation for the state-of-the-art algorithms, as well as all their testing efforts. Finally, Chapter five – *conclusion* – concludes this document.

Chapter 2

Outlook

This chapter discusses the work context and introduces the problem to be solved. We begin with a brief introduction to *real-time systems* particularly focusing on the avionic sector. We follow with a discussion/presentation of the certification standards mandating the development of avionic applications. Finally, the problem is introduced, and the value of its solution is assessed.

2.1 Context

Throughout history, from the stone age, the industrial revolution, to modern day – the information age –, humanity has been on a path of discovery, a quest for knowledge. From the ground to the sky, and finally the stars, humanities achievements stand on the shoulders of key discoveries: fire, the wheel, electricity, and computing. Today, humanity stands at another fulcrum point: automation.

Computers have taken charge of many life-and-death functions: they guide lasers that sculpt the eye, drive autonomous vehicles, and even self-landing rockets. Rigorous, fast and precise, ever more able to perform complex tasks. Many of those in *real time*, where the correctness of applications not only relates to the correct output but also the moment in time it is delivered.

One of the most representative examples of critical real time systems is *avionics*: electronic systems used on aircraft, artificial satellites, and spacecraft. In this context, most systems are *hard real time*. That is, most tasks running in the system are associated a *deadline*: a moment in time where they must be completed. A deadline violation of a *hard real time* system can result in a catastrophic failure where human life or the integrity of the system is at risk.

Examples of these systems include automatic processes, control, monitoring, communication, navigation, weather forecast, autopilot (AP), ground proximity warning systems (EGPWS), and anti-collision systems such as TCAS [4]. The latter, for example, stands for traffic collision avoidance system and forms a digital bubble around the airplane. When two bubbles intercept, the systems communicate and inform the pilots of their relative positions, a procedure known as *TA* (traffic advisory). If the relative distance between two aircrafts is such that it becomes a safety hazard, the systems communicate and agree on a path that ensures that both aircraft escape safely. This procedure is known as resolution advisory (*RA*) and possesses the capability of alerting the pilots or reconfiguring the auto-pilot to execute the computed solution. The aviation sector relies so much on the system that in emergency situations where a contradicting solution is given by the air traffic controller and the TCAS system, pilots are ordered to *disregard* the air traffic controller [3].

In some situations, these systems have the power to *override* the pilot's control input, such as the Airbus fly-by-wire system [74]. Fly-by-wire is an electronic interface that replaces conventional mechanical flight controls. Flight control movements performed by pilots are converted to electric signals and transmitted by wires. In essence, the pilots fly the computers, and in turn, the computers fly the aircraft. Besides increased efficiency, this allows for automatic signals to be sent by computers to help stabilize the airplane, or prevent unsafe operation outside the aircraft's performance envelope.

In other situations, these systems completely replace the human component of the system. Today's airlines fly with just two out of the five original crew members of earlier models [88]. In the 1910s, commercial flight crews were composed of: a (1) Captain and a (2) Flight Officer, the pilots-in-command; a (3) Flight Navigator responsible for accurate navigation through celestial navigation and long-range direction finding land-based stations; a (4) Flight Engineer in control of pressurization, fuel, engine, electric, air conditioning, and hydraulic systems; and a (5) Radio

Operator who handled telegraphic and voice radio communications between aircraft and ground stations. Today, only the pilots-in-command remain, both of which have their jobs greatly simplified. All the remaining crew members – and their functions – were replaced by more reliable and efficient automated systems. "*A pilotless airliner is going to come; it's just a question of when*", said the president and CEO of Boeing Commercial Airlines, James Albaugh, at the AIAA Modeling and Simulation Technologies Conference.

Modern *glass* cockpits are highly automated with digital flight instruments in place of the traditional analog dials and gauges that constituted primordial *steam* cockpits. These flight systems, such as electric engine control (EEC), flight management system (FMS), autothrottle (AT), autopilot (AP), etc., are capable of "*controlling every part of the flight envelope from just after take-off to landing*" [96]. This automation provides many benefits; it has greatly increased safety and comfort while extending weather minima[1]. Indeed, today's state-of-the-art Instrument Landing Systems (ILS), specifically Category IIIc, have no Decision Height (DH)[2], nor Runway Visual Range (RVR) constraints [2], allowing pilots to perform auto-lands in zero visibility. Paradoxically, automation has also introduced a new hazard. Extensive use of automation systems such as ILS can erode the necessary skills to fly the aircraft. This is known as *automation dependency* and has been a factor in many incidents [97, 24].

It becomes evident with these examples, that failure to meet correct functional and timing constraints of these computing systems may cause catastrophic consequences. Such systems are called *safety-critical*. Although these examples are limited to the aviation sector, many more exist in other contexts such as railway, automotive, and nuclear industries. Given the high level of *criticality* of these systems, it is crucial to guarantee that they work correctly.

In recent years, the functionality, efficiency and effectiveness required of computer systems for *safety-critical real time* applications has increased exponentially [10]. Naturally, this led to an explosion of the complexity in these systems [115, 38]. Scaling computing capacity to meet these requirements has become difficult, both the research community and the private sector are investing significant efforts into solving the complex problems associated with high criticality real-time systems.

2.2 Background

Scheduling is a common occurrence in everyday life. Many industries such as retail, healthcare, and construction have some form of scheduling problems. Examples of these problems range from employee tasking, to aircraft landing clearances, including logistic planning, etc. Given its importance and broad applicability, scheduling is one of the most researched topics in literature. The Business Dictionary [27] defines scheduling as: "determining when an activity should start or end, depending on its duration, predecessor activities and relationships, resource availability, and target completion date".

Scheduling is also a core activity of computing. Through *scheduling algorithms*, computers decide what task will be executed next. In real-time systems where strict timing constraints exist, these activities are even more crucial to their operation. Within this domain, there are two system categories [87]: (1) *hard real time*, where a timing constraint (deadline) violation can result in catastrophic consequences; and (2) *soft real time*, where deadline misses cause Quality of Service (QoS) degradation, but are otherwise without any serious consequences.

Traditional avionic architectures, called the *federated* method, were based on stand-alone subsystems interconnected with real-time communication buses where each component was performing a single task [60]. Although reliable, this approach was expensive due to numerous heavy equipment racks required for each and every system, and up to 100 kilometers of cables to interconnect all those systems [46]. The new Integrated Modular Avionics (IMA) architecture combines several, potentially independent, systems on one general purpose computing node, sharing power, memory, network, and computing resources. IMA significantly reduces production and maintenance costs while increasing system reliability, reusability and evolutivity [60]. The first airplane to adopt the IMA concept was the Boeing 777 [81]. Newer civilian and military planes such as the Airbus A380, Boeing 787, Airbus A400M Atlas, and the Lockheed C130 AMP were also developed according to this model.

[1]The minimum required weather conditions under which aviation operations are permitted.

[2]The lowest specified altitude during the approach phase where visual contact with the runway is required, or a missed approach is declared, and go-around procedure is triggered.

Given its high criticality level, avionics are heavily regulated via certification standards. For a program to be used in a *hard real time* safety critical context, it must be *certified*. Certification is a lengthy and expensive process, accounting for a significant portion of the development efforts, in which the program output and time requirements are proven to be correct in *all* circumstances impacting the safety of the system.

The certification requirements placed upon high criticality real-time systems are extremely strict, and the deployed solutions tend to be *simple* so as to facilitate the certification process through which they must pass [42, 32].

As an example of certification cost, over 50% of the software development efforts for the Boeing 777 were in the area of analysis and tests for the purpose of certification [20]. For the Boeing 787 Dreamliner, FAA operatives logged more than 200,000 hours on technical work. Boeing's engineers exceeded that mark while proving compliance with more than 1,500 airworthiness regulations; presenting 4,000 documents comprising of test plans, flight tests, and safety analysis; and demonstrated compliance with over 16,000 federal safety standards relating to inspection and test parts. Boeing's Dreamliner 8 year-long certification process was "*the most rigorous in Boeing's history*" [17].

Example of certification standards include: ISO 26262 [100], IEC 61508 [31], DO-178B [8], DO-254 [6], DO-178C [7] and ARINC [5]. Of those, ARINC 653 and DO-178C specify the latest software and hardware requirements for avionic systems. These certification standards introduce, among others, stringent scheduling constraints to which all systems must be compliant.

ARINC 653 was developed by aviation experts to provide "*the baseline environment for application software used within Integrated Modular Avionics (IMA) and traditional ARINC 700-series avionic systems*", and defines a task scheduling policy.

2.2.1 ARINC 653 System Model

ARINC 653 specifies a system model according to the IMA philosophy [5]. In ARINC 653 avionic systems are grouped in hardware modules powered by multiple processor units. The modules are managed by an ARINC 653 compliant real-time operating system, such as PikeOS [102], VxWorks [45] and LynksOS-178 [104].

Central to ARINC 653 is the concept of *partitioning*. Each hardware module hosts multiple partitions, each composed of one or more processes (figure 2.1). Processes within a system are partitioned with respect to space and time. *Time partitioning* guarantees that the timing characteristics of tasks, such as their worst-case execution times are not affected by the execution of other tasks in other partitions. *Space partitioning* ensures that the data of one partition cannot be corrupted by another partition.

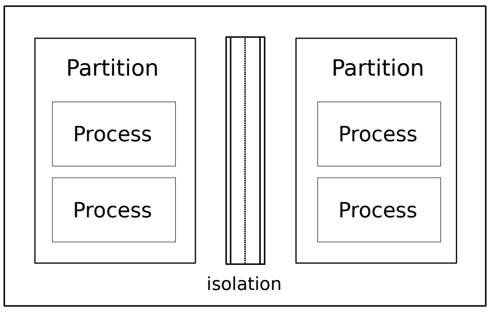


Figure 2.1: ARINC 653 Partition time and space isolation

The underlying architecture of a partition is similar to that of a multitasking application, where processes are isolated and composed of multiple threads. Within a partition the constituent processes operate concurrently, sharing the available services and resources. Each process is uniquely identifiable, with attributes that define its scheduling, synchronization, and overall execution.

Partitions are bound to one processor module during system configuration by the system integrator, as illustrated in figure 2.2. In the current version of the standard, which was written with single core processor in mind, process or partition migration of any kind is not allowed. All processes allocated to one partition must be executed on the same processor. It is expected that this requirement will be updated to allow migrations between cores of the same processor once multicore processors are accepted and integrated into future avionic systems.

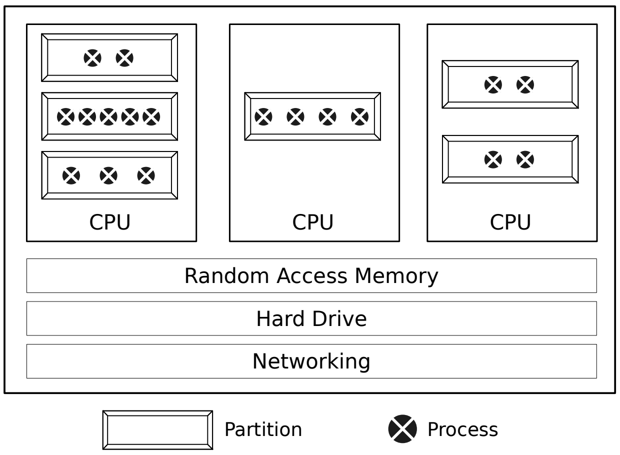


Figure 2.2: ARINC 653 Hardware Module

ARINC defines intra-partition and inter-partition communication through the APEX interface (APplication/EXecutive). Communications are dispatched during partition context switches, if the source and destination partitions do not reside on the same hardware module, the message is sent to the target module via Avionics Full Duplex Switched Ethernet (AFDX).

ARINC 653 Scheduling Policy

In ARINC 653, the scheduling policy is *deterministic* and *hierarchical*. A time-based dispatcher cyclically schedules partitions following a predefined schedule table built by the system integrator. The table defines scheduling intervals allocated to each partition within a *major time frame*. The length of the *major time frame*, i.e., the schedule period, is usually set as the least common multiple of all partition periods in the system. The scheduling table is periodically repeated throughout the system runtime operation. Figure 2.1 and 2.3 contain an example of a scheduling table.

	P	Start	End
1	P_3	0	4
2	P_2	4	9
3	P_1	9	11
4	P_2	11	16
5	P_3	16	19
6	-	19	20

Table 2.1: Example partition table in tabular form

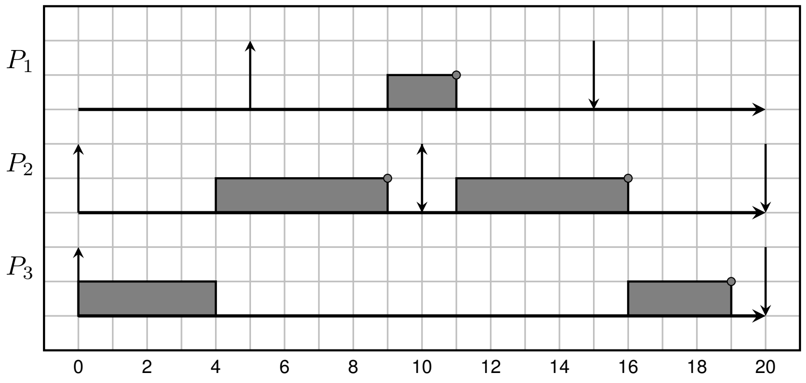


Figure 2.3: Example partition table in diagram form

The schedule table must ensure that the periodic computation demand of all partitions is satisfied by the appropriate size and frequency of the partition windows within the major time frame.

Within a partition, processes are scheduled by a fixed priority preemptive scheduler. The scheduler delegates all partition resources to the highest priority process within the partition. Ties are resolved by First-Come-First-Served (FCFS).

Partitions scheduling characteristics are defined by (1) a unique identifier, (2) a period of execution, and (3) an execution time. Within the partition, processes may be periodic or aperiodic, and are characterized by (1) a unique name, (2) base priority, (3) period, (4) time capacity or Worst Case Execution Time (WCET), (5) a deadline, (6) current priority, and (7) next deadline time.

2.3 Problem Introduction

Cyclic executives are used to schedule safety-critical real-time systems because of their determinism, simplicity, and efficiency. One major challenge of the cyclic executive model is to produce the cyclic scheduling timetable. This problem is related to the bin-packing problem [34] and is *NP-Hard* in the strong sense. Unless P is equal to NP, an exponential amount of work is, in the worst-case, required to generate a schedule.

Currently, the partition scheduling tables are mainly built based on the experience acquired by experts. The ultimate goal of the thesis is to automate this process by developing efficient algorithms that generate close to optimal partition scheduling tables, where optimality is defined as:

1. All partitions timing demands are respected. Each partition receives enough processing time to ensure all its processes complete within their timing constraints.
2. The size of the table is kept sufficiently small, saving valuable memory space.
3. Partition precedence and exclusion constraint are fulfilled.
4. Context switches between partitions are minimized.

Context switches can introduce significant overhead, in IMA cache-related overheads can increase task execution times up to 33% [18]. In addition, since it is at that time instance that inter-partition messages are pushed toward their destination, the cost of context switching can be significant, wasting valuable processing resources.

The best solution would be to develop an algorithm which yields an optimal solution, i.e., the lowest number of context switches and table entries possible while respecting application timing, precedence, and exclusion constraints. If this is not possible in reasonable time, a heuristic algorithm should be developed as it allows for a faster computation of the result. Given the extensive solution space size [114], it is computationally intractable to iteratively search the entire space; however, optimization algorithms can provide a good solution in reasonable time.

The problem in its general form has been significantly studied by the research community. Several works proposed in the literature are aimed at producing a feasible schedule. Some contributions produce sub-optimal results via optimization algorithms or heuristics; others achieve optimality by iterating the solution space with Branch and Bound (B&B) or Mixed Integer Linear Programming (MILP) formulations. To generate a solution, these contributions may require massive amounts of computing power and large systems may be computationally infeasible.

As far as we know, state-of-the-art solutions define optimality as any solution that effectively schedules the entirety of the task set. We extend this definition of optimality by adding one more constraint: a schedule is deemed optimal if for a given task set, the schedule is feasible and the number of context switches is minimized.

By maximizing processor utilization within each computing node, fewer nodes are required to schedule a given task set. For avionics, a competitive industry where a 1% efficiency increase can translate into hundreds of thousands in yearly fuel savings [106], this presents a significant contribution. Further, through those fuel savings, global Carbon Dioxide (CO_2) emissions can be reduced, helping comply with the always more stringent regulation rules defined by the governing authorities and contributing to the well-being of our home planet.

The results of this work are also a significant contribution to scheduling theory. We intend on extracting scheduling properties that can be charted onto into existing optimization algorithms and heuristics, hence guiding their research toward better solutions efficiently. The authors intent on solving broader problems such as combined task and message scheduling in distributed real-time systems would be able to utilize an optimal solution. Currently, the earliest deadline first algorithm is used by some authors to schedule tasks on computing nodes [1].

Our contribution also extends beyond high criticality real-time systems. Many systems providing day-to-day functionality also use cyclic executive [79], from entertainment systems such as toys and audio players, to data acquisition and sensing systems such as environmental systems for temperature monitoring at weather stations, or control systems such as air conditioning and industrial robots. Many of these embedded systems have limited resources, where context switching overheads can be prohibitively expensive. Through our contribution, context switches can be minimized, reducing both energy and consumption costs through reduced CPU usage or the use of less powerful hardware on these platforms.

Chapter 3

State of the Art

This chapter covers a formal review of state-of-the-art schedulers. We begin with an introduction to the cyclic executive model, using online scheduling as an analogy. Lastly, we present published scheduling related work, along with a comparison of these contributions.

3.1 Cyclic Executive

The cyclic executive is well covered in the scientific literature [87, 11, 114, 53, 86, 57, 79]. Although many authors may refer to it by different names: offline scheduling, time-table scheduling, time-line scheduling, time-triggered, static-cyclic, static scheduling, pre-scheduling, and pre-run-time scheduling. In this section, we define the cyclic executive scheduling model and compare its benefits and drawbacks to online scheduling alternatives.

In the cyclic executive, schedules are computed offline, where the characteristics of all system tasks are known in advance. For this reason, it is possible to utilize computationally expensive scheduling algorithms. Proving that a schedule is feasible is simple, by iterating the executive it can be shown that all constraints are met. Once a feasible task sequence is found, the schedule is saved in a static data structure which will be interpreted by the real-time kernel dispatcher – an online scheduler which schedules tasks based on the executive. Because there is no computation at run-time, the dispatcher has $O(1)$ complexity. At its basic form, the dispatcher can be implemented by writing a for-loop which invokes tasks through function calls and manages idle periods using sleeps [79]. Modern systems utilize a table-driven architecture which is loaded during system configuration.

For a dynamic scheduler such as Fixed Priority (FP), the schedule is computed both online and offline. An offline algorithm assigns priorities to tasks based on their requirements, such as the Rate Monotonic (RM) algorithm [63], an optimal policy in the class of Fixed Priority scheduling algorithms for periodic or sporadic tasks where periods are equal to deadlines. The online scheduler may be preemptive, limited-preemptive, or non-preemptive, and generally schedules the highest priority task in the system's ready queue.

3.1.1 Schedulability Analysis

Fixed priority applies utilization based schedulability tests to ensure a given task set is feasible. Liu and Layland proved that for RM, a given task set of n periodic tasks with unique periods is feasible if $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$, where n is the number of tasks in a task set τ, U is a bound on the total system utilization, C_i is the task's computation time, and T_i the task's period. For two tasks $U = 0.8284$, and when n approaches infinity, $U = 0.6931$. Hence even with only two tasks, RM can only schedule up to 80% utilization; when the number of tasks n grows, RM feasibility guarantees drop by $\approx 10\%$ to reach $\approx 69\%$. Cyclic executive can schedule any task set up to 100% utilization if an algorithm exists that can build such schedule. Earliest Deadline First (EDF) [63] is an example of such algorithm with pseudo-polynomial complexity.

Fixed Priority Scheduling algorithms are not optimal, especially in non-preemptive systems, where a lower-priority task τ_i with a longer C_i can cause a higher priority task τ_j to miss its deadline when τ_j is released after τ_i began its computation. In such cases, the system may be schedulable by introducing an idle period until the release of τ_j [70].

Because tasks are always executed in the same order, the cyclic executive is fully deterministic. While it is true that determinism is not necessary to achieve predictability [11], it reduces the total number of system states to one. By executing the system once, it can be proven that no bugs due to task execution order are present. The same cannot be said for fixed priority, which is predictable, but not deterministic, and can have many distinct task execution orders.

3.1.2 Scheduling Constraints

Cyclic executives can consider scheduling constraints such as task precedence and mutual exclusion, arbitrary release times, and low jitter requirements without added runtime complexity or overhead [114]. This is not the case for fixed priority: mutual exclusion and precedence relationships can conflict with the natural priority order of the system, requiring an additional task management layer.

These mechanisms are computationally heavy and can introduce significant jitter in the system. For example, periodic interrupts can delay even the highest priority task in the system, as the underlying operating system moves tasks between its priority queues. Complex real-time mutual exclusion mechanisms such as Priority Ceiling and Priority Inheritance [92], are required to protect shared resources from concurrent access while respecting task computation demand, increasing the time spent on activities which are not part of the real-time systems. Additionally, these mutual exclusion mechanisms must avoid deadlocks during run-time, which requires that they be conservative, resulting in situations where a task is blocked even though it could have proceeded without causing a deadlock [114].

In cyclic executive, mutual exclusion and precedence overheads are handled offline by defining constraints on task pairs. The offline scheduler will generate a schedule compliant with these constraints, performing the same function as an online scheduler with zero run-time overhead.

3.1.3 Efficiency and Scalability

The size of the scheduling table is usually determined by the Least Common Multiple (LCM) of the periods of all tasks. If these are not harmonically related, or are relatively prime, the table size can become quite big. By operating a task on a slightly higher frequency – reducing a task's period –, it is possible to obtain a satisfactory table length at the cost of reducing overall processor utilization [11]. Fixed priority also exhibits a similar problem, when a given task set has many different periods, many priority levels are generally required, which can increase memory requirements and run-time scheduling overhead [93].

The fixed priority model is much more dynamic. It can improve system responsiveness by adapting task execution order such that periodic and sporadic demand is served within a minimum interval. Cyclic Executive models sporadic tasks as periodic tasks by reserving system resources so that sporadic tasks can always be served at their minimum inter-arrival time, which is likely to be an unrealistic scenario. This computational reserve impacts periodic tasks, as resources which could be used to reducing response times are left idle.

Additionally, cyclic executives display high software design fragility, incurring excessive life cycle costs [11]. Any changes to the system, such as adding a new task or updating an existing one, may require the schedule to be updated or rebuilt. To avoid having to test the entire system, system integrators often try to find empty slots or even break functional separation by appending new functionality to existing tasks. In comparison, adding new functionality to FP driven system merely implies applying a schedulability test.

3.1.4 Closing Arguments

In hard real-time systems, the most differentiating criteria is determinism. Cyclic executive imposes strict deterministic execution ordering, greatly simplifying runtime behavior analysis and verification.

The cyclic executive has constant $O(1)$ complexity and can achieve up to 100% utilization. It can solve complex application constraints with constant runtime overhead and minimal jitter. Given its deterministic behavior, verifying these constraints is a straight-forward process.

The cyclic executive's strength is also its weakness. Because the schedule is built offline it is less dynamic than fixed priority, and offers worse responsiveness. System modifications may require the schedule to be rebuilt, resulting in high update/modification costs.

3.2 Related Work

This section presents state-of-the-art contributions proposed in the technical literature. Mono-processor, multi-processor, and heterogeneous scheduling solutions have been included. Although multiple-processor and heterogeneous systems are an extension of the uni-processor problem, the unique and interesting approaches to these systems may nevertheless be applicable to uni-processor.

The state-of-the-art is divided into three classes of algorithms: *exact*, *heuristic*, and *meta-heuristic*. Exact algorithms are optimal and are guaranteed to find a feasible schedule if one exists; they may require significant processing power for large task sets and may be computationally infeasible within an acceptable timeframe. Heuristic algorithms are based on polynomial techniques which may sometimes produce an optimal solution within a set of well-defined constraints. Meta-heuristics are heuristic techniques to efficiently search large solution spaces, but are pseudo-random and probabilistic and cannot guarantee an optimal result. Significant computation time and power may be required for meta-heuristics to produce good results.

The state-of-the-art has been partially developed based on the following state-of-the-art reviews: [22, 116, 53].

3.2.1 Exact Algorithms (EA)

Exact algorithms are optimal, i.e., an optimal solution is guaranteed to be found if one exists. Exact techniques search the entire solution space with exponential time and space complexity. Large task sets require a significant amount of time and processing power and may be computationally infeasible within an acceptable timeframe.

Petri Nets (PN)

Petri nets [77], developed by Petri in his dissertation, are bipartite graphs similar to state transition diagrams with strong mathematical foundations behind them. Petri nets are composed of tokens, places, transitions and arcs. Places may hold one or many tokens; tokens move between places through arcs – roads – linking transitions and places. A time Petri net incorporates the notion of time into a Petri net. The main obstacle to their application to scheduling is the search space size, which grows exponentially in relation to the number of tasks and precedence/exclusion constraints.

An optimal preemptive scheduler based on Petri nets which considers precedence/exclusion constraints was proposed by Barreto [12]. In this contribution, a system model is generated by converting periodic tasks to a Petri net, enumerating all possible task execution orderings, which are then iterated by a depth-first search. To minimise the space size, partial-order reduction pruning – where the activities that can be executed in any order are pruned – is used. Experimental results were performed with non-preemptive real-world task sets. The first, a non-preemptive robotic arm task set with 37 tasks took 7.8ms to execute. The second, a mine drainage system with 10 tasks arrived at a solution after 138 states. For small task sets, this solution is very fast; larger sets may be computationally infeasible due to the exponential complexity of the problem

Another proposal [37] by Grolleau and Choquet-Geniet supports preemption, inclusion and exclusion constraints, reader/writer problems, and synchronous/asynchronous communication. The authors reduce the state space size by removing unnecessary preemptions, using efficient data representations, and pruning infeasible branches. To test performance a 7 task example with 99.4% utilization unschedulable by any online algorithm is made schedulable by this proposal. The authors offer no performance benchmarks for larger task sets.

Branch and Bound (B&B)

First introduced by Land and Doig [55], branch and bound is a tree/graph based transversal algorithm. It iteratively enumerates all candidate solutions by order of smallest cost until a feasible solution is found. Since B&B always explores the branch with the smallest cost, it is guaranteed to find an optimal solution if one exists.

Xu and Parnas developed an offline best-bin-first based scheduler which considers precedence and exclusion constraints. Their proposal supports preemptive task sets and is optimal in the sense that it can find a schedule if one exists. The algorithm divides tasks into segments according to shared resources and begins its search in a root node generated by preemptive EDF; if the schedule is unfeasible, the algorithm locates the first unschedulable section and attempts to shift its execution backward, generating all combinatorial permutations as child nodes. For each child node, EDF is used to produce a schedule. The process continues until a feasible schedule is found, all nodes have been searched, or until all system memory has been exhausted. Runtime tests show the algorithm can generate results for up to 100 segments, but the authors do not specify run-times. Xu extended the model to homogeneous multiprocessor systems [112] but restricts preemptions and task divisions due to complexity constraints.

In [1], Abdelzaher and Shin extend the previous idea of Xu and Parnas to preemptive heterogeneous and distributed systems. The B&B approach solves the combined task and message scheduling with precedence and exclusion constraints. The algorithm performs very well for utilizations up to 80% but requires significantly more time for demanding task sets. To solve this problem, Abdelzaher and Shin introduce a greedy depth-first search that can solve task sets up to 90% utilization within an acceptable time frame.

Jonsson and Shin introduced another B&B based multiprocessor non-preemptive scheduling algorithm [49]. The authors explicitly avoided preemption because "*the presence of non-negligible context switch overhead makes it very hard to find feasible schedules if unconstrained preemption is allowed*". The proposal models the solution space through a search tree, where vertices are scheduled tasks and iterates it by generating all possible permutations; last-in-first-out (LIFO) is used to decide which node to expand first. During each iteration, a lower bound on the total cost is calculated for each vertex; only the vertices capable of producing a better result than the current global best are saved for expansion. To further reduce the search space, solutions that are infeasible are pruned; additionally, commutative task orderings, where the execution order of tasks always yields the same result, are also pruned. Experimental evaluations were performed with 12 to 16 tasks on 2 to 4 processors with a time limit of four hours. Even with these small task sets, the number of searched vertices ranged from 100 to 10000. For larger systems, the optimal approach may be infeasible, and the authors suggest using heuristics to calculate the lower bounds and produce faster results.

Self Adjusting Dynamic Scheduling (SADS) [40] is a centralized non-preemptive scheduling strategy developed by Hamidzadeh and Lilja. SADS reserves a single processor within a multi-core context to schedule the remaining cores and finds the optimal schedule regarding communication and processing costs. SADS first proposal was based on a computationally expensive branch and bound algorithm [56]. Their latest contribution introduces two new heuristics to improve efficiency, *minimum-remaining-execution-time (MESADS)*, and *depth-bound SADS (DBSADS)*. MESADS is an A*[1] monotone[2] heuristic that estimates the execution cost of the entire schedule by assuming tasks execute with minimal expenses. SADS is a depth-first search of the original branch and bound method that prioritizes partial schedules where more tasks have been assigned. DBSADS outperforms SADS and MESADS in performance but does not guarantee optimal solutions.

Mathematical Formulation (MF)

Mathematical formulations model the scheduling problem through rigorous mathematical formalisms and equations. Within this approach, proposed solutions are based on two main families of tools: satisfiability module theories (SAT), and Integer Linear Programming (ILP).

SAT models the problem through mathematical boolean formulas composed of logical connectives such as conjunction, disjunction, and negation [26]; a SAT solver searches the solution space for a solution using efficient propositional logic algorithms. The problem of propositional logic is NP-complete [25].

ILP, or integer linear programming, models the scheduling problem through a mathematical optimization/feasibility problem where all variables are integers. Mixed-integer linear programming is an extension of ILP in which the integer restriction is relaxed, and some variables are allowed to be non-integers. An ILP solver is responsible for taking an ILP formulated problem and producing an answer. Integer programming is NP-hard [25].

In [67], Metzner et al. present an SAT-based approach to the non-preemptive task and message allocation problem in distributed real-time systems that is guaranteed to find an optimal solution. A binary search algorithm is developed on top of the SAT solver to reduce run times. Experimental

[1] Introduced by Hart, Nilsson, and Raphael in [41], A* is an informed search algorithm that extends branch and bound. Like its sibling, A* chooses the path with the smallest cost but considers the one that appears shortest first by estimating the total distance to the desired node. A* is optimal if the heuristic is monotone.

[2] A heuristic is labeled monotone if its cost estimate is always less or equal than the actual cost.

results show that for a task set of 43 tasks, their approach has a run time of 48 mins. The same task set with additional requirements takes 6 hours.

Another SAT solver [109] for scheduling mixed-criticality multi-core applications has been proposed by Voss and Schätz. Their approach generates an optimized assignment of tasks concerning memory, timing, and criticality constraints. The proposed solution can be used to solve problems of 50 tasks in 15 computational nodes in about 4 hours.

An optimal integer linear programming solution [64] has been developed by Mangeruca et al. The authors map mathematical formulas to provide optimum priority and deadline assignments to corresponding ILP formulations. Two algorithms, fixed priority, and EDF were converted to ILP formulations that are optimum under precedence and schedulability constraints. Experimental results were performed with a combustion engine task set with 5 tasks with precedence constraints, but the authors do not disclose run times.

Puffitsch proposed an ILP solution [82] to partition and schedule non-preemptive tasks on many-core platforms. The partitioning and scheduling phases are both performed at the same time, considering memory contention caused by other cores on the computing platform. Additionally, dependencies and message based communication among tasks are supported. To evaluate the solution, both real-life task sets, using flight application software designed by the EADS Astrium Space Transportation, and random task sets were used. For large task-sets comprised of 800 tasks, the solution is capable of finding valid schedules within 45 mins using up to 50GB of memory.

Model Checking (MC)

Model checking is the process of verifying whether a model is compliant with a specification. Cyclic Executive schedulers of this type search the entire solution space until a model compliant schedule is found, or all combinatorial permutations have been evaluated. A multiprocessor non-preemptive cyclic executive scheduler based on model checking has been developed by Ravn and Schoeberl in [85].

In this model, each task is represented as an automaton with idle, ready, running, and error states. Additionally, each task has two local clocks: the first represents the elapsed execution time, and the second the time elapsed during the tasks period. There is no global scheduler, tasks are allowed to transit between periods as long as guards and invariants preventing invalid states, such as missed deadlines, are satisfied.

To test performance, Ravn and Schoeberl used a 16 task workload for an autonomous vehicle. Tasks deadlines and computation times were tuned to accommodate higher utilization tests. For a single-core task set with 82% utilization, a schedule is found within seconds; if the utilization is increased to 97%, a schedule is found in 10 minutes. In a multi-processor environment with 2 heterogeneous CPUs and 134% utilization, the scheduler requires 15 minutes; a slight utilization increase gave no results after an hour.

3.2.2 Heuristic Algorithm (HA)

In ancient Greece, the word *"heuriskein"* was used to express the ability to find, invent and discover [16]. Today, the word heuristic defines an approach to a problem that does not guarantee an optimal result, or any result will be found.

In this section, we present heuristical scheduling algorithms which can generate a schedule in constant or polynomial time complexity. This type of algorithm is widely deployed in real-time systems because of its efficiency, predictability, and analyzability. Depending on their behavior, online scheduling algorithms are be classified as *static/dynamic*, *preemptive/limited-preemptive/non-preemptive*, and *work-conserving/non-work-conserving.*

A *static* scheduling algorithm always assigns the same priority to a given task; Rate Monotonic (RM) is an example of a static fixed priority scheduler that assigns priority to tasks based on their period. A *dynamic* scheduler may assign different priorities to tasks during the execution; Earliest Deadline First is an example of a dynamic priority scheduler where task's priorities are elevated as they close on their deadline.

A *preemptive* algorithm allows a task to be preempted by another higher priority task. In *non-preemptive* executions, tasks execute continuously until their computation is finished. In *limited-preemption*, the scheduler is allowed to postpone a preemption based on some criteria.

If the processor is allowed to idle when there are ready tasks, the scheduler is classified as *non-work-conserving.* A *work-conserving* scheduler never allows the processor to be idle when the ready queue is not empty.

Being heuristical in nature, these algorithms only provide limited schedulability guarantees within a well-defined set of constraints or no guarantees at all. Some algorithms are optimal under very specific constraints, e.g., if periods are harmonic or loose harmonic. A task set is deemed *harmonic* if the periods of its constituent tasks are all multiple of each other; if each period is an integer multiple of the smallest period the task set is classified as *loose harmonic*.

An algorithm's schedulability guarantees are formally proven and well defined mathematical expressions. A *schedulability analysis* determines if a given task set is feasible, that is if all tasks in the system complete within their assigned deadlines. A *response time analysis* calculates the maximum period of elapsed time from the release of a task until its computation is complete. A response time analysis can therefore be used as a schedulability analysis by checking that all task's response times are smaller than their deadlines.

A given schedulability analysis method may be *sufficient* and/or *necessary*. A sufficient schedulability test can only determine if a given task set is schedulable; if the test is false, then no conclusion can be drawn regarding the schedulability of the task set. A necessary schedulability analysis can determine if a task set is unschedulable; it defines a property *necessary* to achieve schedulability, without whom, it is impossible for all tasks to meet their deadlines.

Rate Monotonic (RM)

Rate Monotonic [63], by Liu and Layland, is a static fixed priority assignment algorithm. In RM, priorities are assigned to tasks based on their period, the shorter the period, the higher the task's priority. RM schedulability theory is very complete, a schedulability analysis for independent periodic tasks is provided by Liu and Layland [63], for both periodic and aperiodic tasks in [108], and with synchronization requirements in [94, 61, 101].

RM is widely deployed in real-time systems due to its simple scheduling mechanism, predictability, analyzability and low overhead.

Deadline Monotonic (DM)

Deadline monotonic [63] is a fixed priority algorithm that assigns priority to tasks based on their relative deadlines. The task with the shortest is assigned the highest priority. It is optimal in the class off FP scheduling algorithms when tasks are independent and have deadlines shorter than periods.

Earliest Deadline First (EDF)

EDF, first proposed by Liu and Layland [63] is a dynamic scheduling algorithm that schedules the task which is closest to its deadline. EDF is optimal on preemptive uniprocessors if: (1) all tasks are independent, (2) context switches and preemptions do not incur any cost, (3) tasks worst case execution times are smaller than their deadlines, and (4) a task cannot suspend itself. Within these constraints, if a task set is not schedulable by EDF then it cannot be scheduled by any other scheduling algorithm. EDF runs in polynomial time and can schedule utilizations up to 100%.

Spuri was the first to propose an exact Response Time Analysis (RTA) for EDF [98]. Guan and Yi proposed two RTA methods that significantly improve analysis efficiency.

A schedulability analysis for EDF task sets with arbitrary release offsets called QPA is presented in [117] by Zhang and Burns. QPA greatly reduces computational requirements over previous methods, in over 80,000 schedulable and 60,000 unschedulable task sets 95% of the task sets complete the schedulability test in less than 30 calculations.

Least Laxity First (LLF)

Least Laxity First (LLF) [105] schedules the task with least laxity. Laxity is defined as the difference between the task's deadline and remaining execution time at a given instant of the schedule. Although optimal for uniprocessor systems, the algorithm is impractical because laxity ties result in constant context switching among tied tasks. Oh and Yang, proposed a solution to solve this problem in [76].

Non Preemptive Earliest Deadline First (NP-EDF)

NP-EDF is a nonpreemptive variation of the Earliest Deadline First algorithm. Like EDF, NP-EDF schedules the task which is closest to its deadline. NP-EDF has been shown by Jeffay, Stanat, and Martel [47] to be optimal in the class of work-conserving algorithms for sporadic task sets. The optimality was however disproven by Nasri and Fohler for periodic task sets with offsets or when tasks do not execute for their worst-case execution time [70]. Jeffay, Stanat, and Martel presented a necessary and sufficient schedulability analysis test for periodic tasks with arbitrary release offsets in [48]. Nasri et al. present a sufficient schedulability test for harmonic task sets in [73].

Fixed Preemption Points (FPP)

Developed by Burns [21], FPP defines a non-preemptive execution model where context switch events are solely triggered by a system call invoked by the running task. FPP divides tasks into non-preemptive blocks; if a higher priority task arrives between two preemption points, preemption is delayed until the next preemption point. Because scheduling is decentralized, this approach is also referred as Cooperative scheduling [22].

Preemption Threshold Scheduling (PTS)

A fixed priority scheduling algorithm [110] proposed by Wang and Saksena, where each task is assigned an additional priority called *preemption threshold*. A task can only be preempted if the priority of the arriving task is higher than the threshold of the running task.

Keskin, Bril, and Lukkien provide an exact response time analysis for PTS in [51]. Saksena and Wang proposed an algorithm to calculate task preemption threshold for a given task set [89].

Buttazzo, Bertogna, and Yao compared PTS to FPP in simulated experiments and concluded FPP can achieve higher task set schedulability [22]. Additionally, Buttazzo, Bertogna, and Yao observed PTS has a simple interface and can be implemented with minimal runtime overhead, but that its preemption cost cannot be easily estimated.

Deferred Preemption Scheduling (DPS)

Baruah introduced the concept of *deferring* or delaying preemptions in EDF [13]. Each task in the system defines the longest interval that can be executed non-preemptively while delaying (deferring) the preemption of another task.

There are two models of deferred preemption threshold: floating and activation-triggered. The floating model is implemented at task level, non-preemptive regions are defined by the programmer through system calls with a duration not exceeding a certain threshold. Activation-triggered is implemented at the scheduler level, where non-preemptive regions are triggered by the arrival of a higher priority task.

A feasibility analysis of periodic tasks with floating non-preemptive regions has been developed by Short [95].

According to Buttazzo, Bertogna, and Yao, the performance of DPS is equal to FPS, generating more preemptions and achieving lower schedulability than FPP. In comparison to PTS, the number of preemptions can be better estimated.

Clairvoyant EDF (C-EDF)

Introduced in [30] by Ekelin, Clairvoyant non-preemptive EDF scheduling uses a form of look ahead to determine if an idle period must be scheduled to avoid a longer task executing over the release and subsequent deadline of another task.

Experiments show CEDF can increase schedulability up to 100% when compared with NP-EDF while maintaining algorithmic complexity – $O(n \log n)$ – through lazy evaluation.

Group-Based EDF (Gr-EDF)

Group EDF [62] by Li, Kavi, and Akl dynamically groups tasks based on the closeness of their deadlines and uses Shortest Job First (SJF) to schedule these tasks within the groups. Group EDF can improve schedulability during high utilization and overload conditions.

Precautious Rate Monotonic (P-RM)

An online non-work-conserving non-preemptive scheduling algorithm based on Rate Monotonic (RM) developed by Nasri and Kargahi. Precautious RM [72] increases the schedulability of non-preemptive task sets by adopting an Idle-Time Insertion Policy (IIP). The policy is defined as follows: if scheduling the highest priority task τ_2 would prevent the task smallest period τ_1 from meeting its deadline, an idle period is scheduled until the next release of τ_1. P-RM has $O(1)$ constant complexity.

Nasri and Kargahi [72] proved that scheduling non-preemptive harmonic tasks can be solved optimally in polynomial time by Precautious RM. In addition, Nasri and Kargahi also introduces response time and jitter analysis for P-RM. Finally, in [70], Nasri and Fohler introduce a linear-time sufficient schedulability test in loose-harmonic task sets.

Lazy Precautious Rate Monotonic (LP-RM)

LP-RM, introduced in [72] by Nasri and Kargahi is a variant of Precarious-RM that schedules any lower priority task between two instances of the highest priority task in the task set. Like P-RM, the algorithm has $O(1)$ complexity.

LP-RM can increase response times and reduce schedulability of P-RM on feasible harmonic tasks.

Efficient Precautious Rate Monotonic (EP-RM)

EP-RM by Nasri and Fohler [71], increases schedulability by assigning multiple tasks to priority groups. Within a group tasks are sorted by increasing period; the task with the smallest period is called the representative task, while the remaining tasks are called tail tasks. Tail tasks are allowed to be scheduled if the representative task is scheduled within the same vacant interval.

The problem of assigning tasks to priority groups is a bin-packing problem. Nasri and Fohler present an offline heuristic called *wise fit* to assign tasks to priority groups in polynomial time. EP-RM schedulability analysis (with wise-fit priority assignment) proves EP-RM dominates P-RM, and that feasible loose-harmonic task sets with period ratios larger or equal to 3 are always schedulable.

Critical Time Window-Based Earliest Deadline First (CW-EDF)

In this algorithm, an idle-time is embedded if scheduling the highest priority task will cause a deadline miss for the *next* job of any other task. Nasri and Fohler, the authors of CW-EDF [70], proved that CW-EDF is optimal for harmonic and loose harmonic task set with period ratios larger than 3.

Nasri and Fohler showed the average schedulability ratio on randomly generated periodic task sets is 80%, a significant improvement over the state-of-the-art Precatious-RM (40%), and work-conserving solutions NP-RM, G-EDF, and NP-RM (15%).

3.2.3 Meta-Heuristic Algorithms (MH-A)

A metaheuristic technique is an heuristical algorithm, often nature-inspired, designed to solve combinatorial optimization of NP-complete or NP-hard problems [14]. Metaheuristics efficiently search the solution space to maximize or minimize a given function. They are generally probabilistic or pseudo-random – stochastic – and do not guarantee any results.

Simulated Annealing (SA)

Simulated Annealing [54] is a probabilistic metaheuristic introduced by Laarhoven and Aarts to perform global optimization in vast search spaces. It is based on annealing, where metals such as copper or steel are repeatedly heated and cooled to increase ductility and reduce hardness. During this process, the material becomes more and more malleable as temperatures increase and less so as temperatures decrease. This process is replicated in simulated annealing: when the temperature is very high, the algorithm is allowed to probabilistically search the entire solution space, escaping local minima; as the temperature is reduced the algorithm's search space is restricted to optimize local minima.

Simulated annealing has been used to produce non-preemptive cyclic executives. The first application of SA to the scheduling problem was made by Wellings [111]. This solution focuses on scheduling hard real-time tasks on distributed architectures, including exchanged messages between tasks on different computing nodes. In this algorithm, SA assigns tasks randomly to processor nodes. The algorithm shows good schedulability, for small task sets comprised of 9 tasks and 5 processors it can find an optimum solution, for larger task sets of 42 tasks and 8 processors the algorithm achieves high schedulability.

Burns, Hayes, and Richardson proposed [19], a non-preemptive scheduling framework that models cyclic executive as a vector of process instances. New solutions are generated by randomizing task ordering. To benchmark performance, Burns, Hayes, and Richardson used a case study provided by Rolls-Royce and adjusted clock rates to test low and high system utilizations. In this benchmark, simulated annealing performed better for high utilizations when compared to the shortest deadline first heuristic algorithm at a price of six thousand times the computing time. The annealer runtime does not increase significantly by higher utilizations, and requires 30 mins on average to find a solution on 50 MIPS machine.

Another proposal by DiNatale and Stankovic [28] schedules distributed static systems with periodic tasks. The scheduler considers messaging, exclusion, and precedence constraints while trying to minimize jitter. The SA schedules tasks iteratively, but randomly. The proposal shows good results for large results comprised of 100+ tasks, generating better solutions than EDF in 80 to 90 cycles.

Time-Partitioning Optimization (TPO) [103], by Tamas-Selicean and Pop, considers a mixed criticality environment where safety-critical tasks are scheduled by cyclic executive, and non-safety-critical tasks are scheduled by fixed priority. Their strategy is composed of three steps: first, an initial partition set is generated by a simple, straightforward partitioning scheme; then simulated annealing adjusts partitions slices such that all safety-critical and non-safety-critical tasks are scheduled while unused partition space is maximized. Finally, a list scheduling heuristic is used to determine the scheduling tables. To test their proposal 10 synthetic benchmarks along with 2 real life use cases were used with 3-5 partitions totaling 15 to 53 tasks. Test executions were limited to 120 minutes and showed that for small task sets, solutions are within 1 to 5% of optimality; for larger cases, TPO produces good solutions.

Genetic Algorithms (GA)

Genetic Algorithms [43], introduced by Holland, search the solution space by mimicking the process of evolution and natural selection. In GA, a population of candidate solutions, called individuals, are iteratively combined and mutated, producing new fitter populations. The evolution cycle continues until a stopping criterion, such as elapsed time or a satisfactory solution, is reached.

Nossal developed a preemptive genetic algorithm for scheduling periodic tasks with hard deadlines on multiprocessors considering exclusion and precedence constraints. The scheduling problem is encoded by selecting start times for each task within a feasible interval, which are scheduled by a least-laxity scheduler. Regarding performance, the algorithm suffers from premature convergence to suboptimal solutions and produces poor results.

Ant Colony Optimization (ACO)

Ant Colony Optimization [29], introduced by Dorigo, mimics the complex behavior of ants when discovering the shortest path to food sources. In the natural world, ants wander randomly until a food source is found, immediately returning to their colony while laying pheromone trails. Other ants, drawn to this pheromone, discover the same food source and go back to their colonies, reinforcing the pheromone trail. Because pheromone evaporates, the shortest paths get marched more quickly and thus become more saturated with pheromone, attracting more ants. ACO mimics this process to converge on optimal solutions.

A partitioned scheduler, where each task is exclusively assigned to a processor, has been proposed by Chen and Cheng [23]. The ACO algorithm assigns tasks to processors scheduled using EDF. To optimize the task assignment problem, the ants travel through a utilization chessboard where each column is a processor, and each line is a task. An ant travels across this chessboard in such a way that only one cell is visited for each row, and the accumulative value of the visited cells is no greater than one. A first-fit-decreasing heuristic was developed to guide the ants towards a good solution by attempting to minimize laxity in target processors. Experimental results show this work performs better when compared to a GA, and an integer linear programming approximation algorithm. Run-times are significantly faster than the GA and ILP solutions, on average ILP required 18 hours, GA an hour, and ACO half an hour.

Laalaoui et al. propose a distance function to escape stagnation situations in non-preemptive ACO scheduling solutions. The distance function selects the next task when more than one is available, guaranteeing that when a task has missed its deadline it will be scheduled during the next iteration.

Tabu Search (TS)

Tabu Search [36] by Glover enhances the performance of local search by blacklisting space regions if no improvement move is available – hence becoming tabu.

After their work in [103], Tămaş-Selicean and Pop proposed a tabu search meta-heuristic [107] to schedule mixed-criticality real-time embedded systems on distributed heterogeneous architectures. Called Mixed-Criticality Design Optimization (MCDO), their proposal considers the following: (1) assigning tasks to processors, (2) mapping tasks to partitions, (3) decomposing tasks to lower criticality levels, (4) sizing and sequencing partition slots, (5) non-preemptive schedule tables for each processor, such that development costs are minimized. To determine the non-preemptive schedule tables a list scheduling heuristic within 5% of optimal execution time is used. Unlike TPO, their previous approach, the mapping and partitioning of tasks are mutually considered, yielding substantially better results. Experimental evaluations show that for small sets of three applications with 50 tasks, TPO can produce an optimum solution with a runtime of 8 hours. For larger sets, superb results are nevertheless produced.

Iterative Local Search (ILS)

Pop et al. developed a design optimization framework for multi-cluster embedded real-time systems [80]. They treat several problems: (1) partitioning of system tasks to time-triggered (cyclic executive) and event-triggered (fixed priority), (2) non-preemptive partition and task mapping, and (3) optimizing communication protocol parameters. The initial partitioning and mapping of tasks are made by a list scheduling greedy approach. If the schedule is unfeasible, they apply a graph-based iterative improvement algorithm called PMHeuristic. The heuristic performs changes to the partition and task mapping by selecting the unscheduled process graph which has the maximum delay from its finishing time and its deadline, compiling a list of graph transformations that can reduce this delay, and applying the one that produces its highest reduction. The heuristic terminates if the schedule is feasible, or no improvement is found. Experimental tests show the framework has good performance, delivering high schedulability, and run-times of 2 minutes for 100 processes, and five hours for large sets comprised of 250 processes. Additionally, the tests show PMHeuristic has the highest impact of all the applied heuristics on system schedulability.

3.3 In Review

In this state-of-the-art we have covered three types of algorithms: *exact*, *heuristical*, and *meta-heuristical*.

The branch of exact scheduling algorithms includes a very diverse set of architectures ranging from Petri nets, to branch and bound, SAT/ILP formulations, and model checking solutions. These proposals all have their unique ways of modeling the scheduling problem, and yet, all share the same drawback: they are exponential in relation to time and space. The computational demand to produce a solution increases exponentially regarding the number of tasks and scheduling constraints such that larger task sets become computationally infeasible. This is such a major restriction that most authors either limit their contributions or experimental results to non-preemptive [67, 12, 49, 82], provide no execution time of their benchmarks [64, 113], or significantly reduce the input size to small and manageable task sets [85, 37, 12, 49]. Even then, the process of generating a schedule can be measured in the number of hours. None of the proposed methods attempt to minimize the number of preemptions; some works [113] can be modified to include preemption costs, but even without this constraint, the run-time required to generate a schedule is already very large. With the constraint, the schedulers would be forced to scan larger portions of the exponential solution space, further increasing run-time.

The realm of meta-heuristics include solutions based on simulated annealing, genetic algorithms, ant colony optimizations, tabu search, and iterative local search. Among all these methods, the most effective scheduling heuristics appear to be simulated annealing and ant colony optimization. Meta-heuristic algorithms are stochastic – pseudo-random and probabilistic –, and because of this cannot provide any quality assurances. Two executions of the same input can generate different results ranging in quality and run-time, or fail to find one at all. Nevertheless, these methods are capable of providing optimal solutions for small task sets and excellent solutions for large task sets comprised of 100+ tasks within 30 minutes [111, 23], a significant improvement over exact methods. Unfortunately, the execution times can vary significantly according to the input, and run-times measured in hours are not common. None of the solutions attempt to minimize preemptions, but one of them attempts to minimize jitter [28], occasionally achieving optimal results.

Heuristic approaches are built to be executed online and run in constant or polynomial time. Online scheduling of partitions is not allowed in IMA [5]; nevertheless, these algorithms describe clever heuristics which may be useful in guiding towards an good or optimal schedule by pruning chunks of the solution space. Indeed some authors couple heuristic methods with brute-force solutions to improve efficiency when scanning the solution space [113].

Several heuristic contributions in the state-of-the-art can be used to reduce the number of context switches. In LP-RM [72], Nasri and Kargahi observed that the task with the shortest period in the task set is the most frequently scheduled task in the system, and by extension, the one that generates the most context switches. Nasri and Kargahi suggest running lower priority tasks between two jobs of this task; by scheduling two jobs of the same task at the extreme range of their periodicity, two of its jobs execute back to back, removing a context switch from the system. In P-RM [72], a limited preemption scheduler, Nasri and Kargahi increase the schedulability of a task set by scheduling idle times; this idle-time insertion policy (IIP) can reduce the number of context switches in a system by postponing the execution of a task which would be preempted. In C-EDF [30], by Ekelin, a form of look ahead – clairvoyance – is used to determine if an idle period is necessary to prevent a deadline violation. Lastly, these heuristic approaches are backed by strong mathematical analysis tools such as response time analysis (RTA). RTA can be used to extrapolate slack times, allowing the execution time of a task to be extended past the release of a higher priority task, preventing its preemption without incurring a deadline miss for any task.

As far as we are aware, none of the solutions in the state-of-the-art completely address the problem of producing a feasible schedule where preemptions are minimized within an acceptable time frame. The authors of this thesis intend on developing a solution which addresses this problem with a unique and novel approach where heuristics meet search space algorithms. By extracting system properties that guide exact or meta-heuristic algorithms towards better solutions, the solution space may be sufficiently pruned to make the problem treatable. Some authors have used this approach before, but an exhaustive solution has yet to be made.

Chapter 4

Project Plan

Contains a detailed description of the project's plan.

4.1 Design

The design of the algorithm will consist of three phases. First and foremost is the extraction of system properties that can help guide exact or meta-heuristic algorithms by pruning solutions which can only be sub-optimal. Second phase is a formal analysis of these properties to evaluate if they always lead to an optimal solution, or whether some criteria must be evaluated before applying them. The end-goal of this phase is a set of heuristics which produce very good, if not optimal results within well defined constraints. Finally, the third phase consists of assembling all the heuristics into one algorithm; a state space search algorithm such as B&B, SAT, or MILP may be required. An analysis of the heuristics is required to evaluate which algorithm suits best.

So far, the following properties have been identified:

1. Heuristic algorithms such as EDF can narrow the solution space towards good results.
2. Executing tasks on the edge of their periodicity can join two job releases of the same task into one, removing one preemption from the system.
3. Scheduling idle times through idle-time insertion policies can reduce the number of context switches in the system by postponing the execution of a task which would be preempted.
4. Extending the execution of a running task past the release of a higher a priority task – executing it for as long as possible – may reduce the number of system preemptions. Schedulability analysis tools such as response time analysis can be used to extract task slack times, ensuring no deadlines are missed.

More research is required to collect more properties. Properties can be collected by analyzing manual or automatically generated schedules. Schedules can be generated by one of the state-of-the-art methods, but some modifications are required since these are not designed to minimize the number of preemptions. Additionally, given the extra constraint, these tools may not produce a result within an acceptable time frame.

4.2 Toolset Planning

A reference implementation of this work will be provided in Scala. The application will serve as a test-bed for runtime experiments, where its performance will be compared against the state-of-the-art. The source code will be licensed under a permissive license, allowing its use in closed source commercial applications.

Scala [90] – SCAlable LAnguage – is a multi-paradigm general-purpose programming language which supports both functional and object-oriented programming paradigms. Scala was chosen for its popularity, stability, and ease of development; it is well designed, incorporating a strong static type system, data streams, strong concurrency support, and a syntax which reduces boilerplate code. Most importantly, Scala runs on the Java Virtual Machine (JVM) and permits seamless interoperability with Java, one of the most popular languages in use today. To publish this work, Scala will be compiled to JVM bytecode, and the resulting binaries will be distributed to application developers.

Other languages could have been used, but the authors of this work preferred an object-functional oriented programming language. Within these languages, Rust was also a good candidate, but Scala is a more mature and interoperable platform which most developers will be more familiar with. Additionally, the Rust toolsets such as libraries and Integrated Development Environments (IDEs) are relatively youthful. Other high performing yet more complex languages such as C and C++ were scrubbed over ease of development, expressibility, and simplicity over raw power.

4.2.1 Architecture

The architecture is a direct translation of the mathematical system model presented in section 5.1 - System Model. Given that we have yet to define an algorithm, and that its intricacies can significantly impact the architecture, we cannot provide an exact definition of the architecture we will use. Composed of only eight classes, the real complexity within the implementation lies behind the algorithms used to implement its functionality. Figure 4.1 illustrated a package diagram for the architecture.

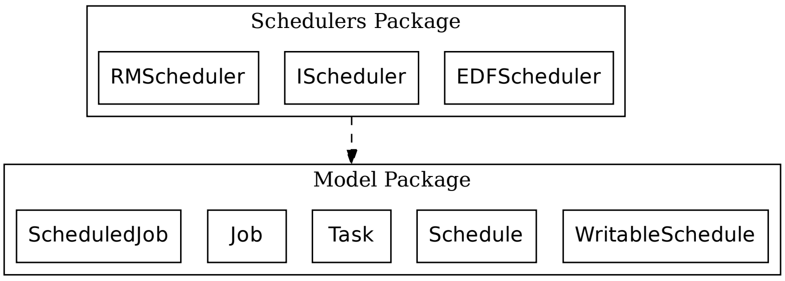


Figure 4.1: A package diagram for the scala implementation.

The architecture deviates from standard event-based schedulers where the scheduler is invoked during the release of a task, or when the running task has finished its computation. Our schedulers are clairvoyant; they have complete and exact system information. Hence, when scheduling a task, they must decide for how long that task will execute for.

To guarantee full interoperability with Java, an object-oriented (OO) *interface* was adopted over a functional one. Object-oriented good practices and design patterns such as SOLID [65], GRASP [58], and Gang of Four [33] were used where applicable.

Figure 4.2 contains a class diagram describing the architecture. Class field visibility symbols denote public '+', protected '*', and private '-' respectively. Note that Scala automatically generates setters and getters for class fields. Scala also utilizes immutability; hence all class public fields are immutable, preventing accidental manipulation of these fields from outside its class space.

Each class has a unique and well-defined role; the architecture exhibits high cohesion and low coupling. *IScheduler* defines a *strategy* of scheduling algorithms and was implemented as a trait with no default methods to guarantee Java interoperability. The trait declares a *generate* method which returns a *Schedule* and is defined by classes which implement this trait.

One alternative to this design would be to model the schedulers as *high order functions*, and defined the IScheduler as a concrete class whose generate method would receive a *higher order function* responsible for scheduling the tasks. Again, this design was disproved due to compatibility issues.

The diagram includes two schedulers: *EDFScheduler* which describes an Earliest Deadline First (EDF) scheduler, and *RMScheduler* which implements Rate Monotonic (RM) scheduler [63]. Each of these classes is a schedule *creator*.

A *WritableSchedule* is a subclass of schedule which grants edit rights to schedules. The goal of this abstraction is to prevent interface pollution on user level by isolating methods which are only useful to schedule building entities in a single hidden interface. WritableSchedule is an *information expert* in regards to scheduling: scheduling algorithms describe similar functions – such as get the highest priority task, or the task with the earliest deadline –, the WritableSchedule aggregates these methods within one class, maximizing code re-usability. Additionally, it also ensures that tasks computational and timing constraints are respected.

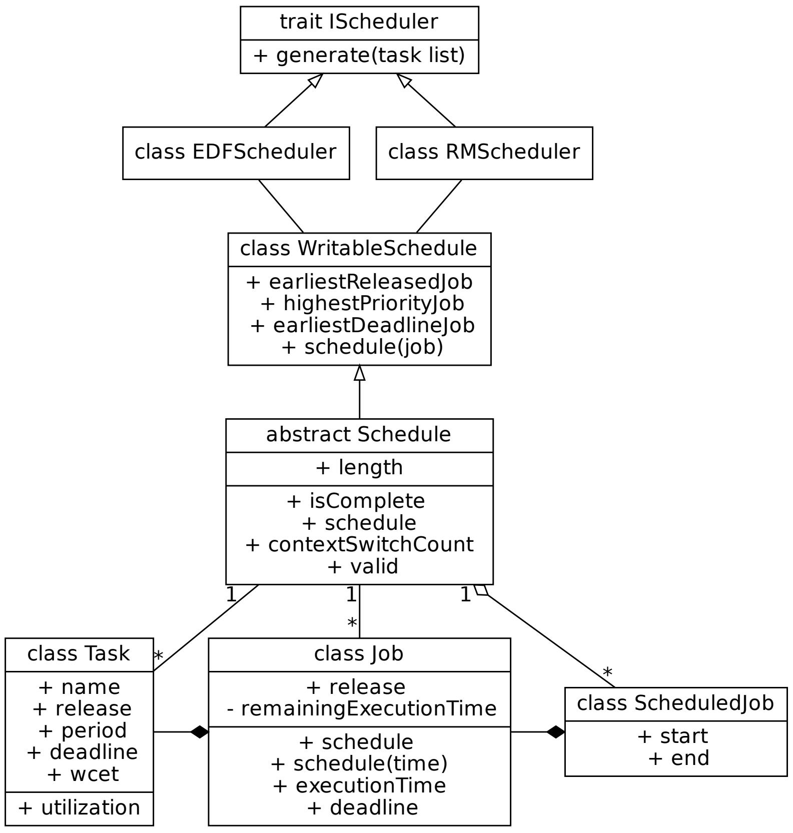


Figure 4.2: A class diagram for the scala implementation.

One alternative to the WritableSchedule would be to design a dedicated factory class responsible for creating and instantiating schedules. We feel the current design strikes a better balance, as we do not need to instantiate an extra class and is ultimately a simpler design.

The schedule object is immutable to userland. It maps the mathematical system model to an object-oriented model. Users define task sets, composed of *Task*s, which are provided to *Scheduler*s and bound to *Schedule*s. From the task set, a list of *Job*s are generated, whose scheduling windows are defined by *ScheduledJob*s.

Call Sequence

To illustrate the call sequence to generate a schedule a sequence diagram is provided in figure 4.3. The user begins by constructing the task set by instantiating tasks. Next, the user aggregates those tasks within a list and invokes the generate method on the EDFScheduler with the list of tasks as a parameter.

The EDFScheduler instantiates a WriteableSchedule, which calculates the length L, and the released jobs within that frame. The EDFScheduler schedules the job with the earliest deadline until all jobs are scheduled. Finally, a Schedule is returned to the user.

4.2.2 Code Quality and Testing

Unit testing will be performed using JUnit [50], a unit testing framework for the Java programming language. Using JUnit allows tests to be carried out from Java, testing the interface's interoperability with it.

All classes will be unit tested. Code coverage will measure the quality of these tests, which will be designed to ensure full code coverage, including hard use cases with high complexity where failure probability is highest.

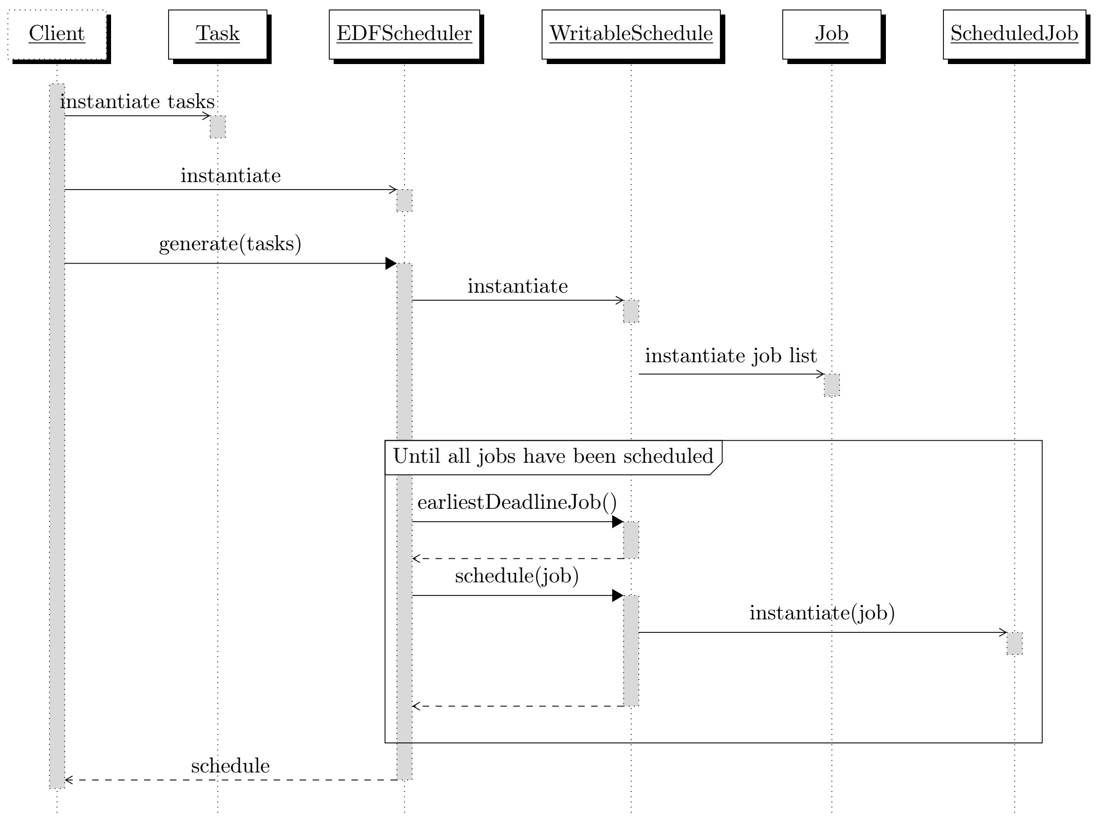


Figure 4.3: A sequence diagram for the scala implementation.

4.3 Schedulability Analysis

We will attempt to devise formal proofs of our algorithm's optimality and provide a Big-O complexity analysis. The algorithm is deemed optimal if it produces a feasible schedule where all tasks timing, exclusion, and precedence constraints are satisfied while the number of context switches is minimized.

If it is not possible to develop formal proofs, statistical tests will be devised where the proposed solution will be compared to the state-of-the-art in regards to the number of context switches with real-world task sets, particular cases, and random task sets generated by the UUniFast algorithm [15] and/or Stafford's randfixedsum [99].

This test consists of proving the hypothesis that, on average, the proposed algorithm generates fewer context switches than another algorithm from the state-of-the-art. The number of context switches is a quantitative discrete variable since, for a given task set, the number is finite and countable. Additionally, because there is a meaningful absolute zero, it is scale ratio data.

We will use the Shapiro-Wilk test to see if our population is parametrical, or in other words evenly distributed. If this is the case both the paired sample t-test and the Wilcoxon signed rank test will be used. Using both tests increases our confidence in the results. However, if the data is not parametrical only the Wilcoxon signed rank test can be used.

4.4 Experimental Results

In addition to building a feasible schedule, it must also do so in a reasonable amount of time. The proposed solution will be compared against the state-of-the-art regarding the computational time required to produce a solution with real-world, particular, and random tasks sets generated by the UUniFast algorithm [15] and/or Stafford's randfixedsum [99]. The running time should be as small as possible; ideally, it would be within minutes, but given the huge solution space this might not be possible. The running time is type discrete continuous since the completion time for a given task set can be half a second. Additionally, because there is a meaningful absolute zero, it is scale ratio data. The same procedure from the schedulability analysis will be used to prove the hypothesis that the proposed algorithm is faster than another from the state-of-the-art.

Chapter 5

A new Scheduling Algorithm

In this chapter a new state-of-the-art scheduling algorithm is introduced.

5.1 System Model

We consider a uniprocessor system executing a set of hard real-time tasks denoted by $\tau : \{\tau_1, ..., \tau_n\}$. Tasks are periodic modeling computations that are cyclically executed.

Each periodic task τ_i in the task set is identified by the tuple (R_i, C_i, T_i, D_i) where $R_i \in \mathbb{N}$ is the first release of τ_i, $C_i \in \mathbb{N}$ is the worst-case execution time (WCET), $T_i \in \mathbb{N}$ is its period, and $D_i \in \mathbb{N}$ is its deadline. In this document, we assume that $C_i \leq D_i$ and $D_i \leq T_i$.

The total utilization contribution of a task, which represents the CPU portion occupied by the task, is given by equation 5.1 [35].

$$U_i = \frac{C_i}{T_i} \tag{5.1}$$

The utilization U of set τ is given by equation 5.2 [35].

$$\forall \tau, U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{5.2}$$

Dependency or exclusion constraints may be defined between tasks. A task τ_i is said to precede another task τ_j, $\tau_i \longmapsto \tau_j$, if τ_j can only begin its computation after τ_i. Task dependencies are transitive, $\tau_1 \longmapsto \tau_2 \wedge \tau_2 \longmapsto \tau_3 \Rightarrow \tau_1 \longmapsto \tau_3$. A task τ_i is said to exclude another task τ_j, $\tau_i \otimes \tau_j$, if no execution of τ_j can occur during the execution of τ_i. That is, τ_j must either complete it's execution before τ_i starts, or τ_j must wait for τ_i to complete its execution before to start its own. Tasks exclusions are symmetric $\tau_i \otimes \tau_j \Leftrightarrow \tau_j \otimes \tau_i$; if during the execution of τ_i, τ_j must be excluded, then the reverse it also true.

Cyclic Executive Model

A cyclic executive schedule ϖ is composed of a set of hard real-time tasks τ, a set of their constraints R, and a set of scheduled jobs executions E. The length of the cyclic executive L, also called *major frame*, is defined by the Least Common Multiple (LCM) of all system tasks.

J is the set of jobs released by all tasks within L, and is denoted by $J : \{J_1, ..., J_n\}$; each job is represented by $J_{n.i}$, where n is the nth release of task τ_i, and is characterized by the tuple ($r_{n.i}$, $d_{n.1}$, c_i), where $r_{n.1} \in \mathbb{N}$ is the release time, $d_{n.i} \in \mathbb{N}$ its deadline and c_i the remaining execution time of J_i which is initialized with C_i. Because tasks are periodic, we have $r_{n.i} = r_{n-1,i} + T_i$; further, $d_{n.i} = r_{n.i} + D_i$. Within the major frame each task τ_i has exactly N_i jobs, as defined by equation 5.3.

$$\forall \tau_i \in \tau, N_i = \frac{L}{T_i} \tag{5.3}$$

The schedule is deemed feasible if, for all tasks in the set τ, each job is scheduled within its predefined time and functional constraints and no two jobs execute at the same time.

The set of restriction constraints R is defined as R: $\{R_1, \tau_1 \otimes \tau_2, ..., R_n\}$. The set of executives (scheduled job executions) E is comprised of scheduled jobs $E^i_j : (s_{j.i}, f_{j.i})$, where i is task τ_i and j the jth job of the task τ_i, $s_{j.i} \in \mathbb{N}$ is the execution start time, and $f_{j.i} \in \mathbb{N}$ is the finish time of that execution.

Example

Let τ be a set of two real time tasks, $\{\tau_1, \tau_2\}$, where $\tau_1 = (0, 1, 5, 5)$, and $\tau_2 = (0, 5, 10, 10)$, with the following restrictions $R = \{\tau_1 \otimes \tau_2\}$. These tasks and restrictions are presented in tabular form on table 5.1 and 5.2, respectively.

τ	R_i	C_i	T_i	D_i
τ_1	0	1	5	5
τ_2	0	5	10	10

Table 5.1: System Tasks

R	τ	*op*	τ
R_1	τ_1	$\otimes$	τ_2

Table 5.2: System Restrictions

J_i	τ_j	r_i	d_i
J_1	τ_1	0	5
J_2	τ_1	5	10
J_3	τ_2	0	10
J_4	τ_1	10	15
J_5	τ_1	15	20
J_6	τ_2	10	20

Table 5.3: System Jobs

The systems major frame, L, is equal to the LCM of (5, 10), which is 10. By equation, 5.3 τ_1 will release two jobs in L, and τ_2 will release one, as demonstrated in table 5.3.

E^i_j	s^i_j	f^i_j
E^1_1	0	1
E^2_1	1	6
E^1_2	6	7

Table 5.4: Example schedule in tabular form

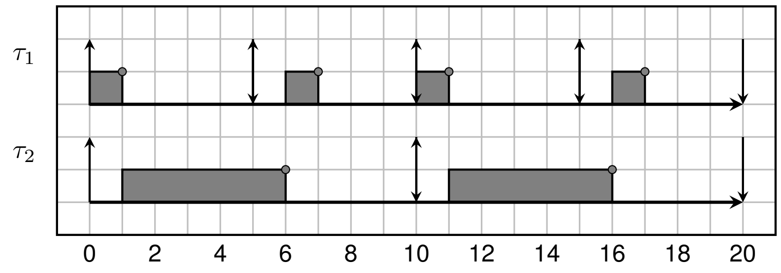


Figure 5.1: Example schedule in diagram form

The schedule, generated using preemptive rate monotonic [63] is represented in table 5.4 and figure 5.1. In Figure 5.1, a task release is represented by $\uparrow$, a task's deadline is displayed by $\downarrow$, and $\circ$ signals the end of computation for that job. The figure represents two cycles of the cyclic executive, as it is twice its length L.

5.2 A new Slack Calculator

Paramount to generating a correct schedule is the concept of *slack*. Slack is defined as the amount of time a job's execution can be postponed without causing its or other job's deadlines to be missed.

The use of an exact and precise algorithm to calculate slack can significantly prune the research space to generate a schedule by cutting invalid branches at their roots, thereby preventing time-consuming look ups that can only produce infeasible solutions.

In this section, we present an exact and precise slack calculating algorithm that is efficient and independent of the system's schedule policy.

To construct such an algorithm, we must determine how much we can delay the execution of a job until its or other deadlines are missed. In addition, we also require the tools to manipulate and update these slack values online as the schedule is being built. We must be able to not only determine if the current schedule is valid, i.e., if the incomplete schedule is a subset of a valid complete schedule; but also be able to manipulate the schedule by replacing jobs or inserting idle-periods; these operations are useful whenever a scheduler wishes to change the order of execution within a schedule.

We will start by explaining how the maximum slack of a job can be compute. We have developed two distinct methods. The first being algorithmic in nature while the second is a formal mathematical approach. Then, we will explain the necessary mechanisms to manipulate slack when constructing a schedule.

5.2.1 Extracting Slack - Algorithmic Approach

Asserting the last instant where a job must begin to execute so that all timing requirements are respected is not a trivial problem. Intuitively, a job should always start executing before or at its deadline minuts its execution time. The slack of that job must therefore be equal to 0 at that instant. However, in most cases, a job may already have a slack of zero well before that instant.For example, a job can have a slack of slack of zero – it must execute promptly upon its release – if it is foreshadowed by a busy period with the release of other shorter-lived or high utilization jobs, such as τ_3 in figure 5.2.

One approach to determine a job's slack is to try postpone its execution until a deadline is missed. Although simple, this method is computationally complex as not only the first, but multiple deadlines can be missed in the future, requiring continuous adjustments to the slacks of all tasks before it. Because there is a strict forward dependency where the knowledge of future releases is required, a smarter approach to the problem of calculating slack is to extract it from the end of the hyper period instead.

Following this method, a scheduler would schedule tasks from the end of the hyper period under latest release and deadline first priority order, implicitly delaying their execution as long as possible. The maximum slack of a job would then be the starting time of its first execution $s_{1.j}$. Named *reverse latest release first* (rLRF) or *reverse earliest deadline first* (rEDF), this scheduler is presented in algorithm 6, along with a complexity analysis is provided in Appendix 7. An example of the rEDF scheduling policy is included in figure 5.2, where it is clear that all jobs are delayed for as long as possible. Table 5.5 contains the latest start time of all jobs since time 0 under column 1 – (A), and the slack relative to the job's release time $Slack_{i,j}$.

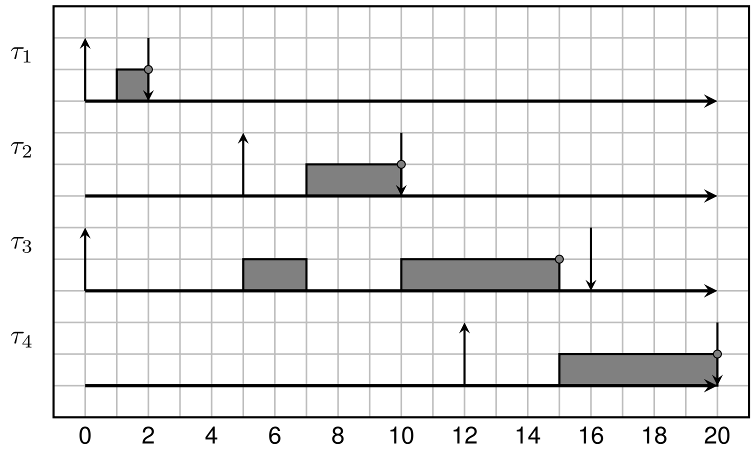


Figure 5.2: An example of reverse EDF scheduling

J	$A_{i.j}$	$Slack_{i,j}$
τ_1	1	1
τ_2	7	2
τ_3	5	5
τ_4	15	3

Table 5.5: The task's slack

During high-load busy periods, a job J_i may have its slack impacted because of: (1) higher priority jobs that must execute within the window of J_i, or (2) lower priority jobs that must be partially executed within the window of J_i. In particular, a job J_i may have a slack of zero, and still, yield the processor to a higher priority release contained within its execution window (1); in other words, having a slack of zero at time T does not mean that the task must execute at time T. Figure 5.2 contains examples of both scenario 1 as job τ_2 and scenario 2 as job τ_4.

For scenario 1, where τ_3 relinquishes the CPU to τ_2 at time 7, the slack of τ_3 is zero and cannot be allowed to become negative. This is because the execution of τ_2 is already accounted for in τ_3's slack. At runtime, given an arbitrary execution order, this scenario may not be easily distinguishable from the one of τ_1 – which is not accounted for in τ_3's slack. To keep the method simple and coherent with the mathematical approach, any job J_i that must execute within the execution window of another job J_j is accounted for in J_j's slack. Hence, the slack of τ_3 must not only account for the execution of τ_2 but also the execution of τ_1, τ_3's slack then becomes 4 instead of 5. During runtime, if a job J_i must execute within the execution window of another job J_j, then the execution of J_i does not decrease the slack of J_j; in this example, the execution of τ_1 and τ_2 do not reduce the slack of τ_3.

Scenario 2 occurs in figure 5.2 due to job τ_4, which must be partially executed in τ_3's window. These cases are intrinsically covered because they can only occur after the first execution of the job whose slack is being computed.

Hence, the slack of a job J can be determined by the start time of its first execution $s_{1.j}$ when scheduled under reverse earliest deadline first, minus its release r_j, minus the execution of any other job that must execute within its release and deadline boundaries that is scheduled to the left of $s_{1.j}$, as formally defined in equation 5.4.

$$\begin{aligned}&\text{let } s_j \text{ be the starting time of the first execution of job J}\\&\text{let } X \equiv \{J_i \ \in J|\ r_i \geq r_j \wedge d_i \leq d_j \wedge s_i < s_j\}\\&\text{then } Slack_j = s_j - r_j - \sum_{J_i \in X(j)} s_i - f_i\end{aligned} \tag{5.4}$$

There are two methods to extract this information: (1) extract the slack from a schedule produced by a reverse earliest deadline scheduler, or (2) extract the slack while simulating a reverse earliest first deadline scheduler.

(1) Extracting the slack from a schedule produced by a reverse earliest first deadline scheduler

Algorithm 1 Computes the slack of a job set by extracting them from a schedule produced by a reverse earliest deadline first scheduler.

```
Require: a schedule produced by reverse earliest deadline first
  function EXTRACT SLACK($jobs$, $schedule$)
    for all $job \leftarrow jobs$ do
      $start \leftarrow$ the start time of the first execution of $job$ in $schedule$
      $slack \leftarrow start - job$.release
      for all $execution \leftarrow schedule$ do
        if $execution$.start $\geq job$.release and $execution$.end $\leq job$.deadline and $execution$.start
  $< start$ then
          $execution\ length \leftarrow execution$.end $-$ $execution$.start
          $slack \leftarrow slack - execution\ length$
        end if
        $job$.slack $\leftarrow slack$
      end for
    end for
  end function
```

Algorithm 1 has a time complexity of $O(J \times (E))$, where E is the number of executions in the schedule and J is the total number of jobs in the task set. This is easily determined because, in order to extract the slack for each job J, two iterations of the rEDF schedule are required: one to find the first execution of job J, plus another to find any executions to its left that are contained within the execution window of J.

However, if we consider the cost of generating the reverse edf schedule using algorithm 6 in Appendix A, then the final complexity is $O(J \times \log_J + E \times (T) + J \times (E))$ since we must generate the schedule via reverse EDF first with a cost of $J \times \log_J + E \times (T)$ [1].

Although inefficient, the algorithm is simple and nevertheless useful to verify more complex algorithms.

(2) Extracting the slack while simulating a reverse earliest first deadline scheduler

Algorithm 2 computes slack exactly like a reverse EDF scheduler would produce a schedule. By merging the steps of generating a reverse EDF schedule and the steps required to extract the slack, it is possible to achieve a time complexity that is asymptotically closer to $O(E)$.

[1]Appendix 7 contains a detailed complexity for rEDF in algorithm 6.

The algorithm ”schedules“ jobs from the end of the hyper period to the beginning, ”scheduling“ the job with the latest deadline that has the earliest release. Once a job J is completed, its slack is computed as its release time minus the start time of its first execution $s_{1,j}$. The job J is appended to a list of jobs that are completed, but still released; if a job whose release and deadline bounds are contained within J's is executed, this execution period is subtracted from J's slack.

Algorithm 2 Computes the slack through a modified reverse earliest deadline first scheduler.

```
function COMPUTE SLACK(task set)
    job list ← task set.jobs
    sort job list by (job ⇒ −job.deadline, −job.release, job.task.id)    ▷ Sort in Reverse EDF
    finished jobs ← empty list
    time ← job list.first.deadline    ▷ Schedule from the end of the hyper period
    repeat
        time ← min(head job.deadline, time) ▷ Determine the earliest time there are active jobs
        job ← job list.iterator
                .takeWhile(_.deadline ≥ time)
                .maxBy(job ⇒ job.release, − job.deadline)    ▷ Get the active job with the latest
release, using earliest deadline as a tie breaker

        runtime ← job list.iterator
                .dropWhile(_.deadline ≥ time)
                .takeWhile(_.deadline > time − job.execution)
                .find(iJob ⇒ iJob.release > job.release or iJob.release = job.release and
iJob.deadline < job.deadline)
                .getOrElse(min(time − _.deadline, job.execution), job.execution)    ▷ Calculate
runtime when a higher priority job is released within the execution of Job

        time ← time − runtime
        job.schedule(runtime)
        if job.finished then
            job.slack ← time − job.release    ▷ Set the job's slack to time minus the job's release
            job.execution ← job.task.wcet
            remove job from job list
            add job to finished jobs
        end if
        for all completed job ← finished jobs do    ▷ Update the slack of completed jobs if the
completed job is within their release and deadline bounds
            if completed job.release > time then
                remove completed job from finished jobs
            else if job.release ≤ completed job. release and job.deadline ≤ completed job.deadline
then
                completed job.slack ← completed job.slack −runtime
            end if
        end for
    until job list.empty
end function
```

The algorithm has a complexity of $O(J \times \log_J + E \times (T + T))$, where J is the total number of jobs, E the number of executions, and T is the number tasks in the system. $J \times \log_J$ is the price of sorting the job list, and $E \times (T)$ is the cost of, for each scheduled execution, resolving the highest priority job and if its execution is interrupted with the release of a higher priority job.

Algorithm 2 could have a slightly improved average time complexity by moving released jobs to a sorted tree, improving the complexity of finding the latest release from linear to logarithmic. However, it would deteriorate the worst-case complexity, since in the worst case all tasks are released at the same time, and all of them need to be sorted in $T \times \log_T$.

5.2.2 Extracting Slack - Formal Approach

The formal method produces the same slack values through a mathematical approach. The method is based on understanding how a job J_i can have its slack reduced. There are two possibilities, (1) because of *body* jobs or (2) because of *carry out* jobs.

A Body job J_j reduces the slack of another job J_i because it *must* execute within the release and deadline bounds of J_i, that is, $r_j \geq r_i \wedge d_j \leq d_i$ (see Fig. 5.3) . The combined execution of all jobs within the window of a job J_i is known as *body workload* W^b.

Carry out slack, or *carry out workload* W^{co}, is caused by a job J_j whose deadlines are later than the deadline of J_i but whose execution *must* overlap with the execution window of J_i (see Figure 5.4). A job J_j *may* generate carry-out slack when: $r_j > r_i \wedge r_j < d_i \wedge d_i < d_j$.

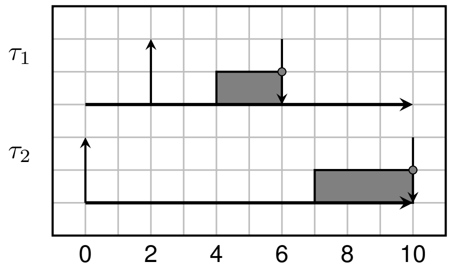


Figure 5.3: An example of body workload generated by τ_1 on τ_2

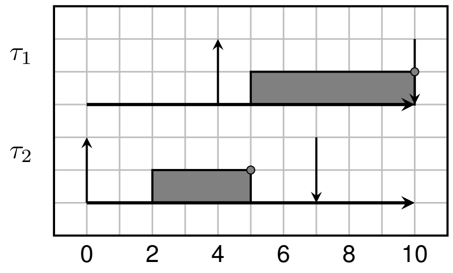


Figure 5.4: An example of carry-out workload generated by τ_1 on τ_2.

Figure 5.3 includes an example of W^b, where the slack of τ_2 is reduced from 7 to 5 since τ_1 is within its execution window and has a worst case execution time of 2, and τ_1 has a slack of 2 since that is the computational requirement of τ_1. An example of W^{co} is provided in figure 5.4, where τ_2 has a slack of 2 because τ_1 pushes its execution backward, and τ_1 has a slack of 1.

To determine the body workload W^b, we need to ascertain how many releases of a task τ are within the window of execution of a job J. This information can be provided by dividing the length of the execution window of J by the period of τ. However, because the first release of τ may not align perfectly with J's, we need to consider the offset between the first release of task τ with that of job J. In addition, a task's deadline can be shorter than its period; this can be considered by subtracting the difference between D_k and T_j of τ_j from the execution window once since this scenario occurs at most one time within the window of any job. Finally, because only the jobs that have both their released and deadline within J's execution window need be considered, we take the floor of the result of the operations described above.

Condensing all the previous steps leads to equation 5.5, which computes the W^b of a job J_i, where T_i is the period of the job whose slack is being calculated, $o_{j,i}$ is the offset of the first release of task τ_j in relation to job i, and d_j and T_j and C_j are the deadline, and period, and wcet of τ_j, respectively.

$$W^b_{J_i} = \sum_{\tau_j \in \tau}^{\tau_j \neq \tau_i} \left\lfloor \frac{T_i - o_{j,i} + T_j - D_j}{T_j} \right\rfloor \times C_j \qquad (5.5)$$
$$o_{j,i} = F_r(j,i) - r_j$$

To compute $o_{j,i}$, one needs to know the first release of task τ_j within a window of execution of J_i. We note that instant $F_{\tau_{j.i}}$, and we compute it with equation 5.6, which returns R_j if the very first job of τ_j is released after J_i's release, or calculates the nth release of τ_j if it is not.

$$F_r(j \in \tau, i \in J) = \begin{cases} \left\lceil \frac{r_i - R_j}{T_j} \right\rceil \times T_j + R_j & r_i > R_j \\ R_j & r_i \leq R_j \end{cases} \qquad (5.6)$$

To determine the carry out workload W^{co}, we must know which job J_j, if any, pushes J_i the most. This can be computed by determining how much a job J_j interweaves with J_i's execution window and subtracting the time J_j's execution can be delayed. Following this reasoning, equation 5.7 computes the W^{co} of a job, where r and d are the release and deadline of a job.

$$\text{let } X(i) \equiv \{J_j \mid r_i < r_j < d_i < d_j\} \text{ then } W^{co}_{J_i} = \max\left\{ \max_{J_j \in X(i)} \{d_i - r_j - Slack_j\}, 0 \right\} \tag{5.7}$$

Note that each task τ_j in τ, may have at most one job in $X(i)$. This job is the last job tau_j released in the execution window of J_i. Hence, it is convenient to be able to compute the last deadline of a task τ_j within a window of execution of a job, denoted by J_i $L_d(\tau_j, J_i)$, which can be computed by equation 5.8.

$$L_d(j \in \tau, i \in J) = \begin{cases} \left\lceil \frac{d_i - R_j}{T_j} \right\rceil \times T_j + R_j - D_j & d_i \geq R_j \\ \varnothing & d_i < R_j \end{cases} \tag{5.8}$$

A higher bound on the slack of a job can be computed with equation 5.9; because equation 5.7 requires the slack of all potential carry out candidates, equation 5.9 is recursive. However, if the slack computation is done in rEDF order, it is guaranteed that all jobs that may cause carry out workload on J_i have been processed before J_i.

$$Slack_{J_i} \geq D_i - C_i - W^b_{J_i} - W^{co}_{J_i} \tag{5.9}$$

Although equation 5.9 is correct, because equation 5.5 *shadows* idle periods that may occur within the window of a job – where it is not possible to execute anything – the value of X^{co} may prove to be erroneous, and by extension, equation 5.9 is incorrect. An example of this is included in figure 5.3, where the slack of τ_2 is equal the execution of $\tau_1 + \tau_2 = 5$, however, within τ_2, there is computation that must begin to execute at time 4 at the latest, which is 1 time unit before 5. The problem is illustrated in figures 5.5 and 5.6, where figure one accurately describes the slack of τ_2 and figure two describes slack as seen by its numerical value of 5.

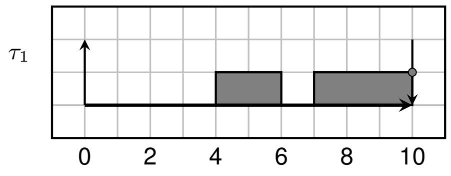


Figure 5.5: Slack without idle-period shadowing

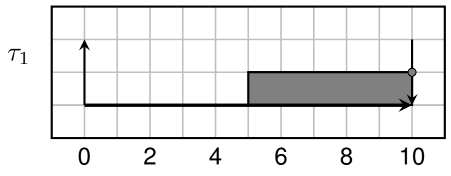


Figure 5.6: Slack with idle-period shadowing

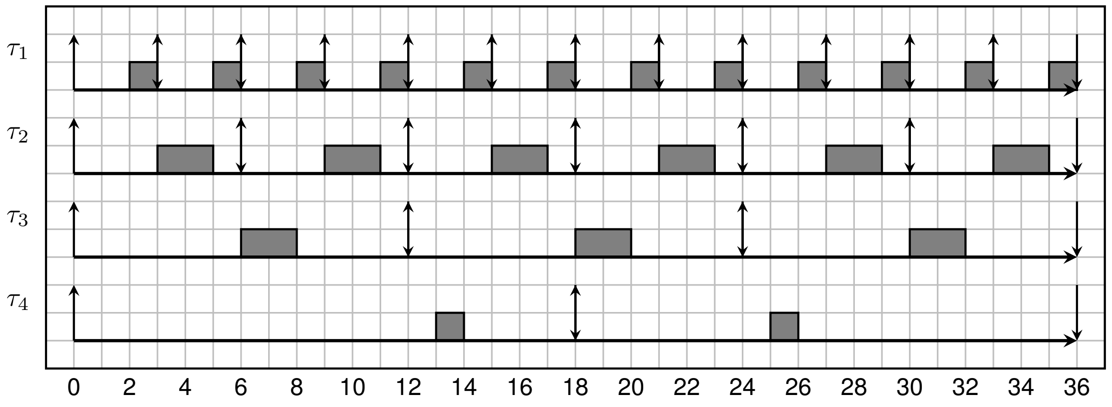


Figure 5.7: An example task set scheduled under rEDF priority order.

Another problem of equation 5.9 is that the some jobs can be accounted twice, once in equations 5.5, and once in equation 5.7. This is a problem because it produces a lower-bound on the slack of a task. An example of this phenomenon is exemplified in figure 5.7, where the second release of τ_3 is carried out by the second release of τ_4; both jobs sharing two releases of τ_1 and one release of τ_2 within their execution windows. In addition, the reverse slack of this release of τ_3 by τ_4 also suffers from the idle-time shadowing phenomenon explained earlier, as there is nothing to execute at time 24, but nevertheless more workload within the execution window of τ_4.

$$\text{let } X(i) \equiv \{J_j \mid r_i < r_j < d_i < d_j\} \text{ then } W^{co}_{J_i}(J_j) = d_i - r_j - Slack_j \tag{5.10}$$

$$W^b_{J_i} = \sum_{\tau_j \in \tau}^{\tau_j \neq \tau_i} \left\lfloor \frac{T_i - o_{j,i} + T_j - D_j}{T_j} \right\rfloor \times C_j \tag{5.11}$$

$$W^{bco}_{J_i} = \min_{J_k \in X(i)} \left\{ \sum_{J_J \in J}^{J \neq J_i} \left(\min \left\{ \left\lceil \frac{r_k - r_i - o_{j,i}}{T_j} \right\rceil, \left\lfloor \frac{T_i - o_{j,i} - D_j + T_j}{T_j} \right\rfloor \right\} \times C_j \right) - W^{co}_{J_i}(J_k) \right\} \tag{5.12}$$

$$Slack_{J_i} = D_i - C_i - \begin{cases} \min\left\{W^b_{J_i}, W^{bco}_{J_i}\right\} & \exists\ J_i \in X(j) : W^{co}_{J_i}(J_j) > 0 \\ W^b_{J_i} & \nexists\ J_j \in X(j) : W^{co}_{J_i}(J_j) > 0 \end{cases} \tag{5.13}$$

To prevent multiple jobs from being accounted twice, the window of execution of a job J_i must be reduced such that it no longer interweaves with its the carry-out job J_j, if such job exists. To fix the idle-time shadowing issue, the W^{co} and W^b must be computed for all jobs that J_i carries out, instead of just the one that it apparently carries out the most; in addition, we must also consider the case where the overall slack is less if we do not consider the W^{co} workload. Such is the case in example 5.7, where ignoring the second release of τ_4 produces the correct slack value for the second release of τ_3.

The final equations to compute the slack are presented throughout equations 5.10 to 5.13. Equation 5.10 computes the carry out workload on a job J_i caused by job J_j; unlike the previous equation 5.7, this one will return a negative integer if the workload extends past its deadline. Equation 5.11 computes thed body workload of a job, ignoring any carry out workload.

Equation 5.12 computes the total workload impacting the slack of a job J_i, eliminating the pitfalls of previous equations. By taking the minimum amount of releases between the execution window of $J_i \setminus J_k$ – excluding the workload that is already considered in the carry out job J_k –, and the complete window of J_i, the minimum operator effectively excludes jobs which are only partially within the window of J_i, i.e., such as those that extend past its deadline. Notice the ceiling operator being used during the calculation of $J_i \setminus J_k$ window; this is to include jobs that bridge from J_i's window to J_k's window.

Finally, equation 5.13 computes the slack of a job J_i by taking the minimum of its W^b and W^{bco} workload, or by W^b if no carry out jobs exist. The equation has a worst-case time complexity of $O(J \times (T^2 + T))$, where J and T are the number of system jobs and tasks, since, for each job J_i, and for each task in the system τ_k, we must compute the W^{bco} of J_i assuming it carries out τ_k with a cost of $O(T^2)$, plus one more iteration of all system tasks assuming that no carry out job J_k exists.

5.2.3 Online Slack Manipulation

As time passes and jobs are scheduled, the slack of all jobs changes must be updated to remain coherent. At runtime, jobs can be in one of three states, pending – unreleased, active – released, or completed – finished.

The slack of an active or pending job J_j is reduced when, within its execution window, an action which is not considered in its slack occurs. This can be one of two cases: (1) an idle period is scheduled, or (2) a job J_i which is not accounted for in J_j's slack is scheduled. As defined in the previous sub-sections, all jobs within the execution window of a job J_j, expressed mathematically as $\{J_i \mid r_i \geq r_j \wedge d_i \leq d_j\}$, are accounted for in its slack. Hence, all the jobs which are not within this group and are scheduled while J_j is in active state must reduce J_j's slack. As proved in theorem 1, these jobs J_i are defined as any job $J_i \in \{J_i \mid r_i < r_j \vee d_i > d_j\}$ that nevertheless interweaves with J_j.

Jobs J_i which reduce the slack of J_j 1. *Let $J_i, J_j \in J$*
Let $J_i \notin \{J_i \mid r_i \geq r_j \wedge d_i \leq d_j\}$
Then $J_i \in \{J_i \mid \neg\ r_i \geq r_j \wedge d_i \leq d_j\}$
$\neg\ r_i \geq r_j \wedge d_i \leq d_j \equiv r_i < r_j \vee d_i > d_j$

Then $J_i \in \{J_i \mid r_i < r_j \vee d_i > d_j\}$

Determining the interweave or overlap O between a job and an execution window starting at s and ending at f can be determined with equation 5.14. In the interest of abbreviation, the equation also contains alternate definitions to calculate overlap between: a job and an execution window, between two jobs, and between an execution and a job.

$$\begin{aligned}
&O(s_i \in \mathbb{N}, f_i \in \mathbb{N}, s_j \in \mathbb{N}, f_j \in \mathbb{N}) = \max\{0, \min\{f_i, f_j\} - \max\{s_i, s_j\}\} \\
&O(i \in J, s \in \mathbb{N}, f \in \mathbb{N}) = O(r_i, d_i, s, f) \\
&O(i \in J, j \in J) = O(r_i, d_j, r_j, d_j h) \\
&O(J_i \in J, E \in E) = O(s_{r_i}, f_{d_i}, s_e, f_e)
\end{aligned} \tag{5.14}$$

Slack manipulation is comprised of the following write operations: idle time scheduling, job scheduling, job de-scheduling, job re-scheduling, and job rollback, as defined below; where the last 4 are exclusive rewrite operations. In addition, it is possible to verify if a schedule is valid, i.e., if it is subset of a complete and valid schedule; to extract the maximum run-time of a job at time T, such that all jobs meet their timing requirements; and to compute the maximum idle period at a given time.

For all these operations, it is only required to iterate at most one interweaving job of each task τ, be it an active job if one exists, a pending release, or none at all when all jobs of τ have been processed. This is based on the tautology that if a scheduling move is valid for the current active/pending release of a job, then it does not impact its next sequenced job; since doing so would imply an invalid move where the slack of its previous release would become negative. The corollary of this tautology is that all operations defined here can be performed in $O(T)$, where T is the number of system tasks.

Determining if a schedule is valid

A schedule is valid if no job exists with negative slack. Indeed, if a job has a negative slack, then by definition of the slack, this job cannot complete by its deadline. This is expressed mathematically as a property in equation 5.15.

$$\forall J_i \in J, \forall t >= 0 : Slack_i(t) \geq 0 \tag{5.15}$$

Extracting the maximum runtime of a job

Using the notion of slack and valid schedule defined above, we can now easily compute the maximum amount of time a job can execute without making any other job miss its deadline. This value depends on the slack of all the other jobs active at the time t. More specifically, the maximum runtime $R^{max}_{J_i}(t)$ of a job J_i at time t, is equal to the minimum slack of all active jobs below its priority order. In other words, the maximum runtime of a job is equal to the minimum slack of all released higher priority – earlier deadline – jobs, as defined in equation 5.16.

$$\begin{aligned}
&\text{let } Z(i) \equiv \{J_j \mid c_j > 0 \wedge O(i,j) > 0 \wedge d_j < d_i\} \\
&\text{then } R^{max}_{J_i}(t) = \min_{J_j \in Z(i)} \max\{r_j - t, 0\} + Slack_j
\end{aligned} \tag{5.16}$$

Computing the maximum idle period

Similarly to the maximum execution time of a job at an instant t, we can compute the maximum amount of time the processor can remain idle from time t onward without having any job missing its deadline. The maximum idle period at time t, I^{max}_t, is equal the minimum slack of all active jobs, as defined in equation 5.17.

$$\begin{aligned}
&\text{let } (i \in P) \equiv \{J_j \mid c_j > 0 \wedge (r_j < t \wedge d_j > t \vee r_j \geq t)\} \\
&\text{then } I^{max}(t) = \min_{J_j \in P(t)} \max\{t - r_j, 0\} + Slack_j
\end{aligned} \tag{5.17}$$

Scheduling a job

To schedule a job J_i at time t for a duration N, the slack of all active jobs J_j that interweave with J_i where J_i is not accounted for in J_j's slack must have their slack reduced by N as described in equation 5.18.

$$\begin{aligned} &\text{let } U(i \in J) \equiv \{J_s \mid c_s > 0 \wedge O(i,t,t+N) > 0 \wedge (r_i < r_j \vee d_i > d_j)\} \\ &\text{then } \forall\ J_j\ in\ U(i) : Slack_j = \begin{cases} Slack_j - N & r_j \leq t \\ Slack_j - (t + N - r_j) & r_j > t \end{cases} \end{aligned} \tag{5.18}$$

Scheduling an idle period

To schedule an idle period with a duration of N at time t, all active jobs that overlap with that period must have their slack reduced in equal measure to their overlap with the idle period being scheduled, as defined in equation 5.19.

$$\begin{aligned} &\text{let } E(i \in J) \equiv \{J_j \mid c_j > 0 \wedge (r_j < t \wedge d_j > t \vee r_j \geq t \wedge r_j < t + N)\} \\ &\text{then } \forall J_i \in E(i) : Slack_i = Slack_i - O(i,t,t+N) \end{aligned} \tag{5.19}$$

De-scheduling a job

When a job J_i is de-scheduled upon a decision made by the scheduler, its execution is replaced by idle time, i.e, it is removed from the schedule. Hence if J_i did not reduce the slack of another active job J_j, the slack of J_j must be reduced by the duration of E_i, as defined in equation 5.20.

$$\begin{aligned} &\text{let } A(i \in J) \equiv \{J_j \mid c_j > 0 \wedge O(i,j) > 0 \wedge r_i \geq r_j \wedge d_i \leq d_j\} \\ &\text{then } \forall J_j \in A(i) : Slack_j = Slack_j - f_i - s_i \end{aligned} \tag{5.20}$$

Rescheduling a job

Rescheduled is defined as moving one execution from one place to another. When a job J_i is rescheduled, the slack impact on another active job J_j is equal to the difference between the overlap of the old and the new executions, E_o and E_n, in relation to the execution window of J_j, if J_i is not accounted for in J_j's slack. The operation is defined mathematically in equation 5.21.

$$\begin{aligned} &\text{let } X(i \in J) \equiv \{J_s \mid c_s > 0 \wedge O(J_i, J_j) > 0 \wedge (r_i < r_j \vee d_i > d_j)\} \\ &\text{then } \forall\ J_j\ in\ X(i) : Slack_j = Slack_j - O(j, E_o) - O(j, E_n) \end{aligned} \tag{5.21}$$

Rolling back a job

This operation is similar to un-scheduling a job except that instead of replacing the execution E with an idle period, its effects are canceled as if the execution E never occurred in the first place. This operation effectively allows a scheduler to "roll back time", as repeated invocations cause completed jobs to be moved to the active and pending states.

$$\begin{aligned} &let\ Y(i \in J) \equiv \{J_j \mid c_j > 0 \wedge O(i,j) > 0 \wedge (r_i < r_j \vee d_i > d_j)\} \\ &then\ \forall J_j \in Y(i) : Slack_j = Slack_j + O(j, s_e, f_e) \end{aligned} \tag{5.22}$$

5.3 A new Scheduler

Determining the execution order of a given non-preemptive task set such that all tasks are scheduled non-preemptively is not a trivial problem. Even more complex is its preemptive equivalent, where not only the optimal execution order must be found, but also the duration of each execution that composes that order.

The problem can be seen as finding the largest continuous regions to fit a given job, while considering all other jobs have the same requirement, such that a global optimum is found where the total number of context switches between jobs is minimized. Because the execution window of a job can interweave with that of other jobs, the number of permutations of possible execution orderings and durations quickly grows, producing a highly intractable research space.

This high complexity does not, on its own, imply that there cannot be an algorithm capable of efficiently navigating the search space to find an optimal, or a good approximate, solution within a reasonable amount of time. The algorithm defined here, is our attempt at such a task.

At a high level, the main reasoning behind our algorithm is to, at any time t, produce the optimal solution up to t in regard to the number of system preemptions. As jobs are scheduled, t increases, and the window of execution grows. Given the newly available space, the schedule may no longer be optimal, and a transformation may be required to bring the schedule back to a near optimal state. By starting with a small window whose optimal solution can more easily be established, and expanding that window iteratively, the efficiency/time with which a good solution can be found increases.

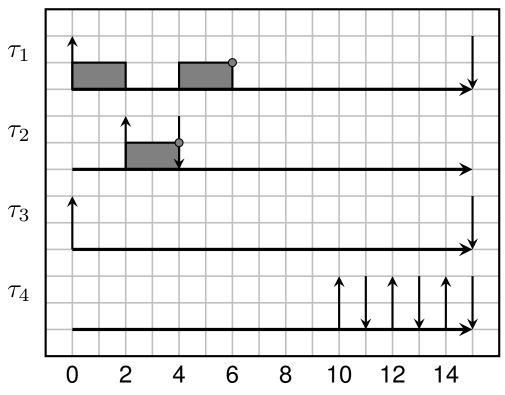


Table 5.6: Example task set - 1

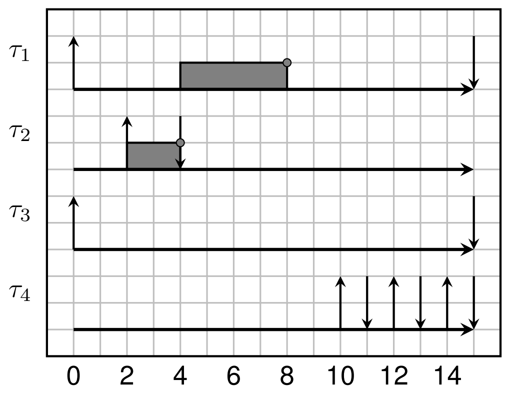


Table 5.7: Example task set - 2

To illustrate this logic, an example task set is scheduled in figure 5.6 to 5.9. The task set is composed of four tasks, where τ_1 and τ_3 have a worst case execution time of 4, and τ_2 and τ_4 a wcet equal to their execution window. In figure 5.6, the taskset is scheduled up to time 6, and at this time the scheduler is only allowed to play with an execution window of 6. By looking at the figure, it is clear that this is the optimal solution within a window execution of that size, as τ_2 must execute from 2 to 4, hence preempting τ_1. As the execution window increases to 8, a preemption can be removed from the system by joining the two executions of τ_1 at time [4,8]. In addition, because of the slack, we know this schedule is a subset of a valid and complete schedule where the computational and timing requirements of all jobs are satisfied. The resulting schedule is illustrated in figure 5.7.

By postponing the execution of τ_1 by 2 time units, the slack of τ_3 is reduced to zero at time 8, meaning that τ_3 has to execute immediately or it will miss its deadline bound. This leaves only one valid execution order, as all active or pending jobs have a slack of 0; this ordering is shown in figure 5.8.

Unfortunately, the previous postponement of τ_1 by 2 time units pushes the execution of τ_3 into a preemption zone generated by multiple releases of τ_4. Optimal for a window of 0-8, the postponement is no longer so when the window of execution is extended to the entire schedule length (or major frame) of 15. A transformation has to be applied to pull τ_3 out of the preemption zone by reverting τ_1's postponement; this lowers the total number of context switches from 8 to 7 – which the optimal number for this job set –, leading to the result illustrated in figure 5.9.

An interesting property of this approach is its inherent greediness. Faced with the choice of scheduling another task, or reducing a preemption by inserting an idle period, the scheduler took the one that reduced the total number of system preemptions that can nevertheless produce a valid schedule. Unlike short sighted greedy algorithms, this approach is recoverable; the algorithm may detect that the previous local optimum is not part of the global optimum and reverts the transformation.

Naturally, the optimality of the solution depends on how each scenario is treated. Many of which may not be easily defined or recoverable in the event they are not part of the global optimum. While we do not claim the solution proposed here is optimal, we do believe it is an approximate of such solution.

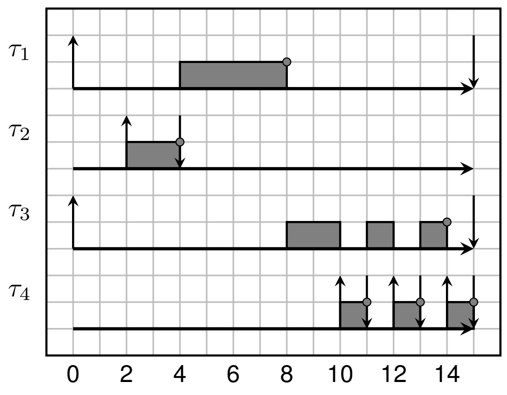


Figure 5.8: Example task set - 3

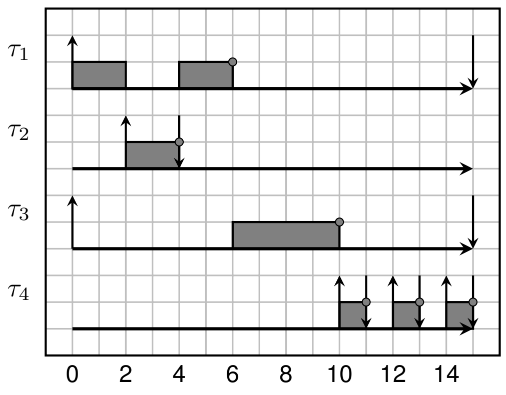


Figure 5.9: Example task set - 4

Algorithm 3 The skeleton of the scheduling algorithm.

```
function SCHEDULE($taskset$)
    while schedule is not complete do
        if there are no active tasks then
            defragment the schedule
        else if can apply a transformation then
            apply a transformation
        else if can complete a job before any higher priority job is released then
            complete the job with the largest execution time
        else
            schedule the earliest deadline job until a higher priority job is released
        end if
    end while
end function
```

5.3.1 Algorithm Structure

A skeleton of our algorithm is presented in Algorithm 3, where upon the existence of active jobs in the current iteration, a transformation that reduces the overall number of system preemptions is applied, if possible. A transformation can include a combination of many actions, such as replacing, shrinking or extending jobs (see detailed descriptions of those actions below). When a transformation is not applicable, the longest task that can be completed is scheduled. This is another example of a greedy and recoverable action; the largest jobs in the system are generally the most difficult to schedule without preemptions which is why they are prefered avor other potentially more urgent jobs (i.e., jobs with smaller deadlines). However, if the arbitrary choice of execution the job with largest execution time leads to unnecessary preemptions later in the schedule, that decision can easily be changed for smaller tasks through one of the schedule transformations mentioned above.

When the release of a high priority – early deadline – job is imminent, making the completion of any job is impossible, the active job with the earliest deadline is scheduled until the release of that job. In these scenarios, it is unclear which job should be scheduled, and transformations may replace the job at a later time if a better fit discovered in the future.

When no active jobs exist, the scheduler attempts to make the best use of this idle period by defragmenting any remaining preemptions in the schedule so far. This is an expensive operation that attempts to combine execution pairs of each job using brute force. Given that the schedule policies presented here already produce a well-defragmented schedule, it is expected this operation will be computationally feasible within a short amount of time.

5.3.2 Schedule Transformations

As presented in Algorithm 3, the schedule may go through several transformations to reduce the number of context switches. We defined 8 different transformations presented in this sub-section. These transformations are applied in a specific order which will be presented later in Section 5.3.4.

(1) Relocate and Schedule Forward

This transformation joins multiple preemptions of a job J_i as it is completed, replacing them with idle periods – hence being classified as an idle time policy. Considering Algorithm 3, these preemptions occur when it is not possible to complete or extend J_i and higher priority jobs are released. Equation 5.19, defined in the slack mechanism chapter, should be used to ensure the idle time does not cause any job to miss its deadline. The transformation is demonstrated in figures 5.10 and 5.11.

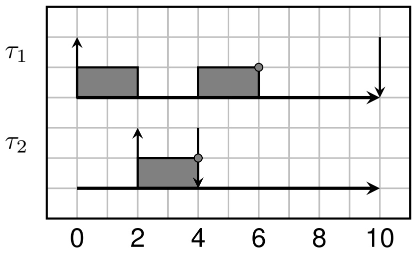


Figure 5.10: Before transformation 1 is applied

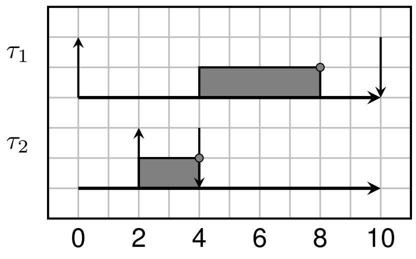


Figure 5.11: After transformation 1 is applied

(2) Reduce preemptions by splitting a job

In this transformation, multiple preemptions of a job J_i are merged by splitting another job into two, thereby creating enough space to prevent some preemptions of J_i. It is the inverse action of transformation 5.3.2. Critical to the transformation is that the gain outweighs the loss, in other words, the global number of system preemptions is reduced after applying the transformation. Figures 5.12 and 5.13 contain a demonstration of the transformation, which is the same example as the second transformation (figures 5.6 and 5.7) in the introductory chapter page 36, i.e, the first job of tau_1 is split in two so as to create enough space to execute tau_3 without preemptions.

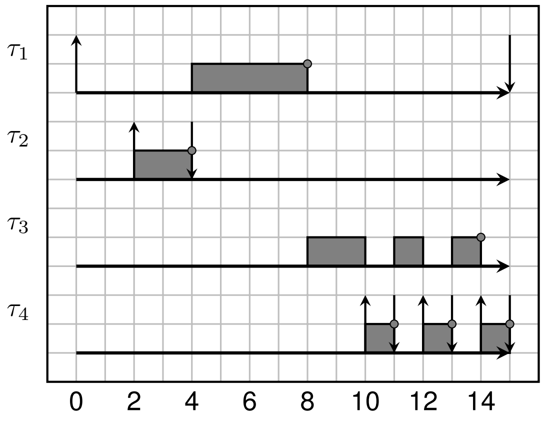


Figure 5.12: Before transformation 2 is applied

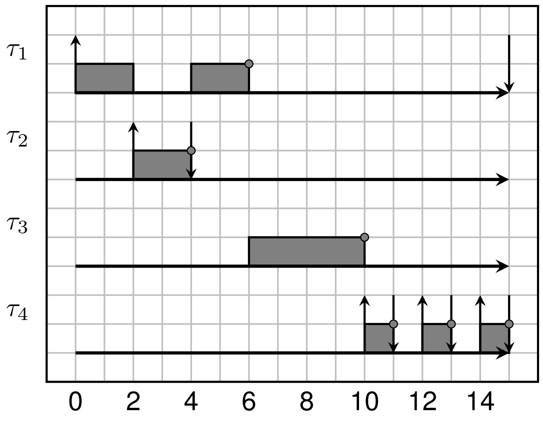


Figure 5.13: After transformation 2 is applied

(3) Extend the execution of a job

One of the primary policies is, on the condition that no jobs can be finished, preempt the running task when a higher priority job is released. This generates unnecessary preemptions when the lower priority task is resumed after the high priority task finishes, yet we could have continued running before the release of the higher priority task. In these conditions, transformation 3 extends the execution of the lower priority task until the release of a higher priority job. Jobs should be extended by last executed first order by iterating the active jobs queue.

The transformation is illustrated in figures 5.14 and 5.15. Figure 5.14 contains the schedule before it is transformed, where τ_3 is preempted by the higher priority release of τ_1, followed by the completion of τ_2 and another preemption of τ_3. Yet this execution of τ_3 can be joined with the first, removing one preemption from the system, as demonstrated in figure 5.15.

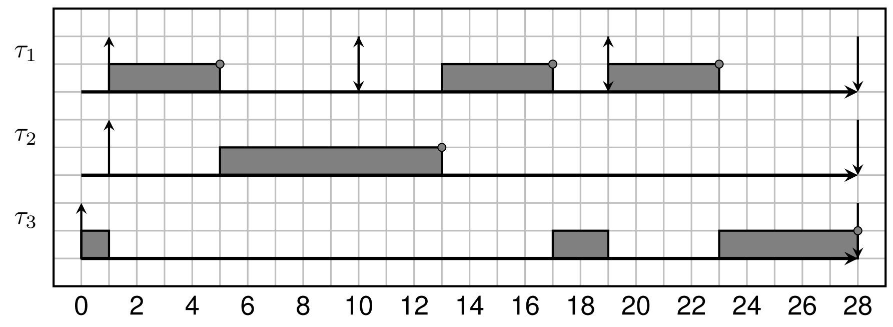


Figure 5.14: Before transformation 3 is applied

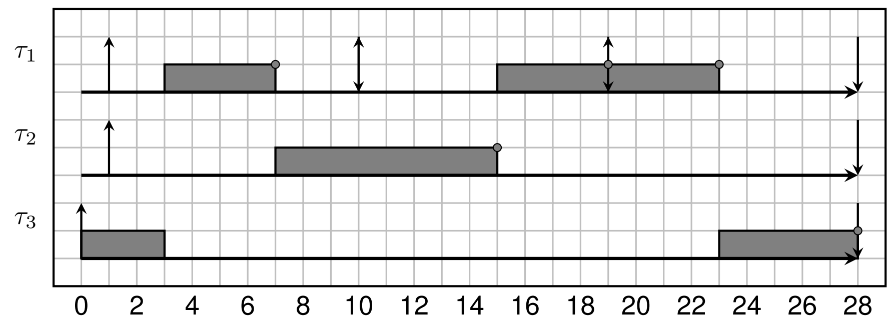


Figure 5.15: After transformation 3 is applied

(4) Complete a job by shrinking a preemption

This transformation is the recovery action of transformation 3, which can push a job into a preemption zone by overextending an execution. The transformation shrinks one or more executions to create the necessary space to schedule jobs which are currently within a preemption zone, reducing the total number of system preemptions. To detect which jobs can be shrunk, the scheduler should keep a list of *extensions* performed by transformation 3. It is illustrated on figures 5.16 and 5.17, where the sub-schedule from [0-19] is equal to the previous transformation described in subsection (3). In figure 5.16, where the first execution of τ_1 is extended from [0,1] to [0,3], τ_4 is pushed into a preemption zone, thereby increasing the total number of systems preemptions. This is solved in figure 5.17 by shrinking the first execution block of τ_1, creating enough space to fit τ_4 in a preemption-free zone at no cost.

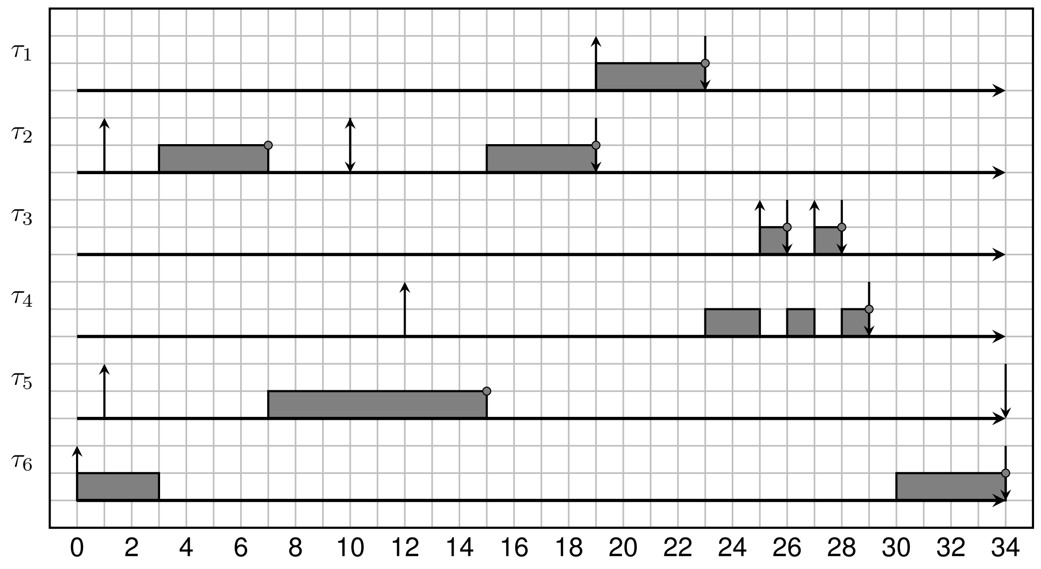


Figure 5.16: Before transformation 4 is applied

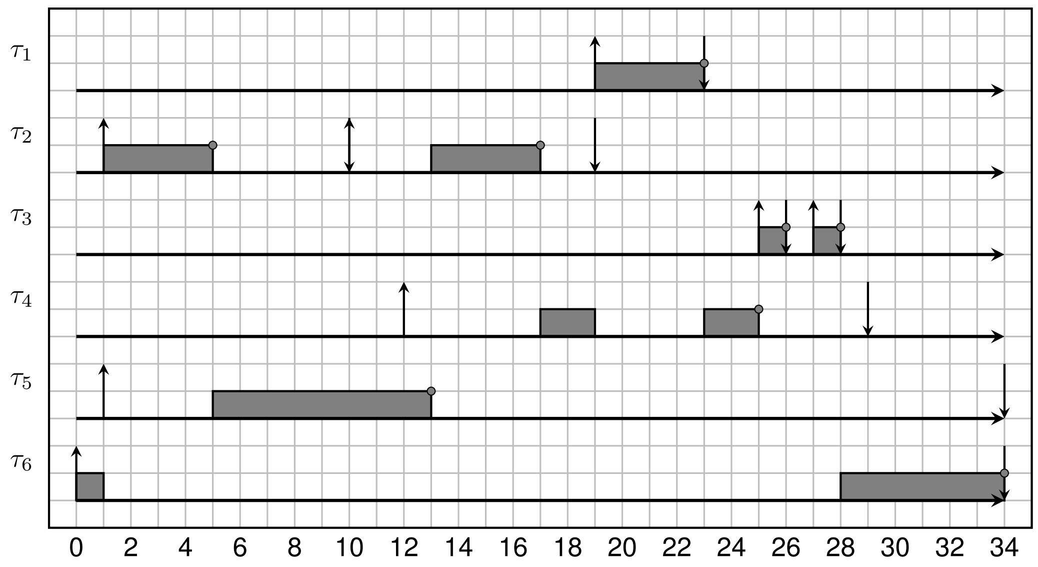


Figure 5.17: After transformation 4 is applied

(5) Knapsack Jobs to prevent preemptions induced by very large tasks

When no transformation can be applied, a core scheduling policy is to, schedule the longest job that can be finished. Given an upcoming release of a higher priority job, continuously scheduling the longest job can exhaust the execution window such that no job can fit in the remaining space. In these scenarios, it's possible to make optimal usage of the execution window by replacing larger jobs with smaller jobs. This is effectively a knapsack problem, as given a window of size N, the goal is to find the number of jobs whose execution sum is total or less than N. The zone to apply the knapsack algorithm is defined as the starts at the previous deadline and ends at the deadline of the next release.

An example is provided in figures 5.18 and 5.19; where on the first figure τ_4 is scheduled first, exhausting 4 time units of the execution window with a dimension of 5, because there is no job with a wcet of 1, τ_4 is a poor choice of task to schedule within this window. A better choice would be a combination of tasks such that the sum of total execution equals 5; as demonstrated in the second figure, this can be accomplished with τ_1 and τ_3.

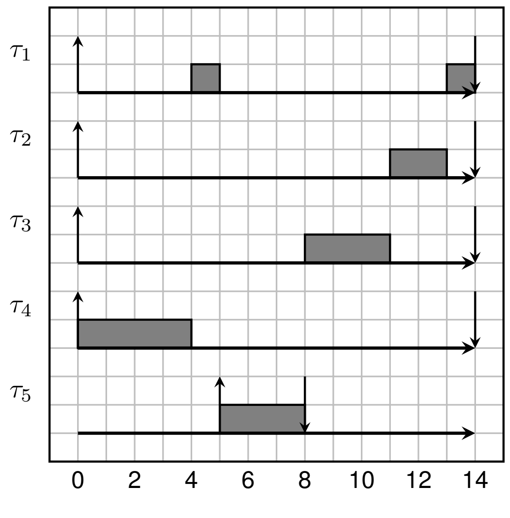


Figure 5.18: Before transformation 5 is applied

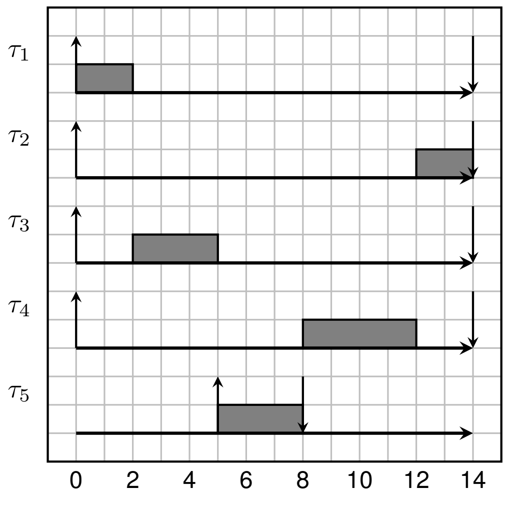


Figure 5.19: After transformation 5 is applied

(6) Swap jobs when at least one can be scheduled non preemptively

In this transformation, two jobs swap their execution so that at least one is completed non preemptively. The scenario occurs when a completed job is a perfect fit to the current execution window, while no active job is. To decide which job J_j to swap for the perfect fitting job J_i, the one with neighboring executions to J_i should be chosen, as these will be joined into one execution block. An example of this transformation is included in figures 5.20 and 5.21, where the execution of τ_2 is exchanged for τ_3 when, at time 14, it becomes clear the remaining window has a size equal to that of τ_2's execution.

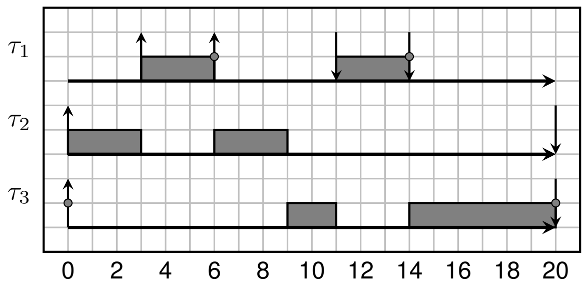


Figure 5.20: Before transformation 6 is applied

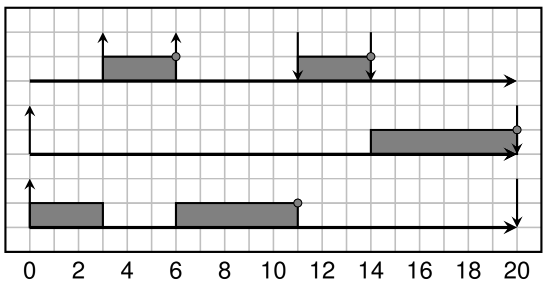


Figure 5.21: After transformation 6 is applied

(7) Create the necessary space to remove a job out of a preemption zone by moving jobs which have a later deadline

In this transformation a job J_i is extracted from a preemption zone by taking the place of other jobs which can be scheduled non preemptively past the deadline of J_i. An example is provided in figures 5.22 and 5.23; where on the first figure τ_6 is preempted into three executions starting at time 14, and ending at 19. Two of these three preemptions can be joined by taking the position of τ_2 at no cost, since τ_2 can be moved past τ_6's deadlines to [18,20].

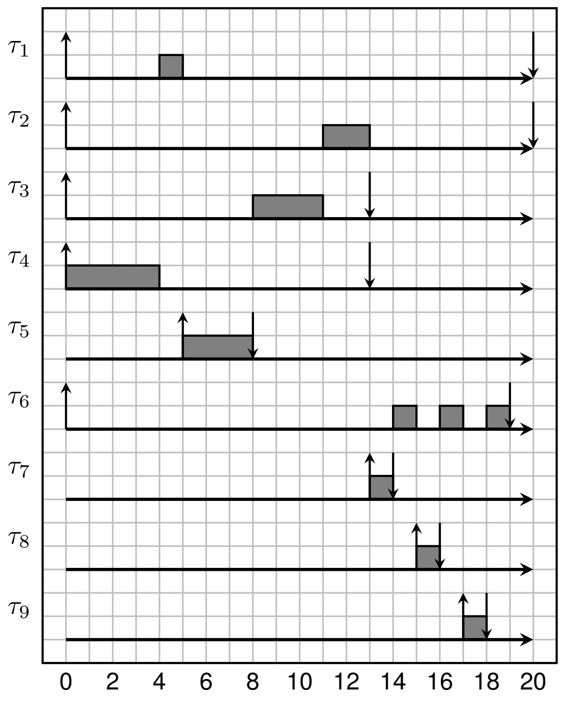


Figure 5.22: Before transformation 7 is applied

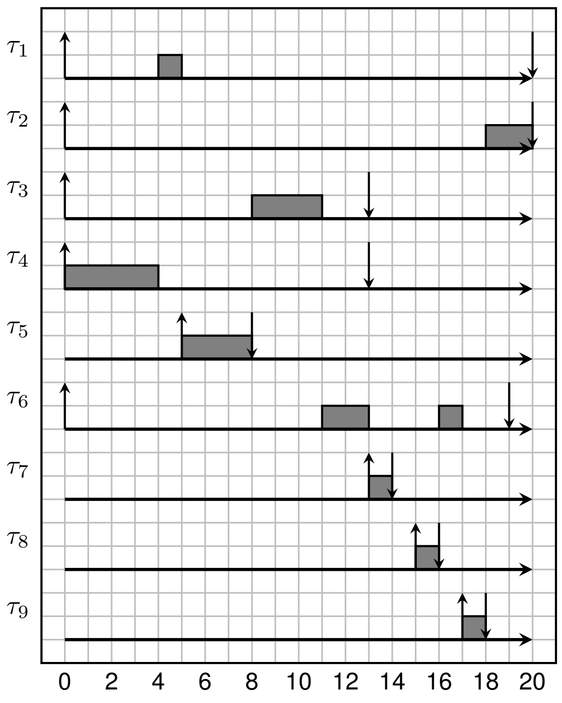


Figure 5.23: After transformation 7 is applied

5.3.3 Schedule Defragmentation

The final transformation, triggered whenever an idle period occurs, is essentially an exhaustive schedule defragmenter. The transformation attempts to make the best usage of the idle period, either directly, by moving jobs into the idle period, or indirectly, by pushing other executions into the period, thereby creating the necessary space to schedule other jobs non-preemptively when the deadlines of these jobs precedes the idle period.

Unlike all previous transformations, which would only execute once per iteration, this transformation will continuously attempt to merge all preemptions of a job until the job is scheduled non-preemptively or all combinations of 2 execution blocks have been checked. The transformation does not attempt to merge executions that cannot possibly be executed non-preemptively, i.e., those whose execution is twice the length of the smallest period in the system, minus the required execution within that period.

A pseudo-code for the algorithm is included in Algorithm 4, where for each job J_i that was released up to twice the longest period ago, an attempt to combine execution pairs of J_i is made by scanning the valid execution window of the job, and attempting to schedule it iterably between each execution block:

1. directly, when enough space exists
2. by shifting other jobs left and/or right at that index, some jobs may have their execution relocated/exchanged with the surrounding jobs if they breach their execution window
3. by relocating the left or/and the right execution

One example, for each case, is illustrated through figures 5.24 to 5.29. An illustration for case 1 is provided in figures 5.24 and 5.25, where two preemptions of τ_1 are joined by moving two executions into an idle period. For case 2, an illustration is provided in figures 5.26 and 5.27, where τ_4 is shifted 4 time units to the right to fit τ_3. Finally, figures 5.28 and 5.29 contain an illustration of case 3, where τ_1 is relocated forward to fit τ_3.

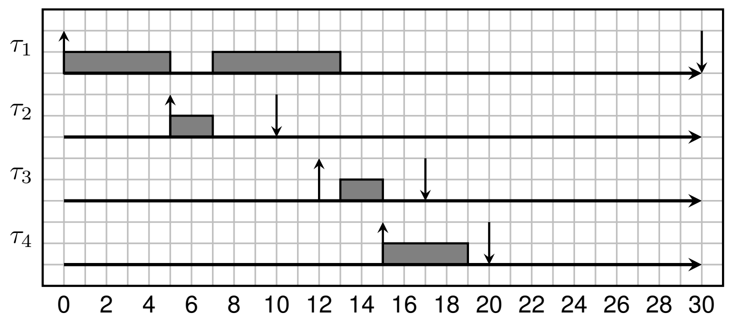


Figure 5.24: Before transformation 8-1 is applied

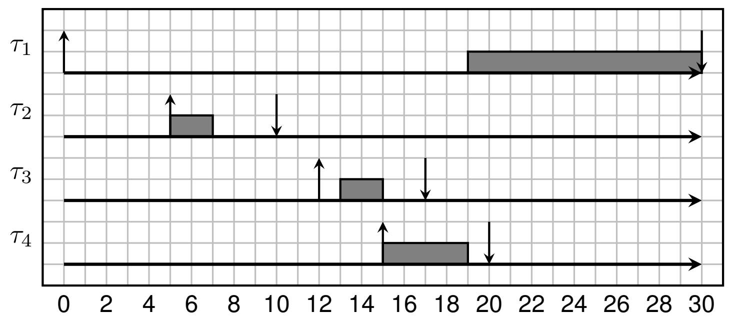


Figure 5.25: After transformation 8-1 is applied

Algorithm 4 The Defragmentation transformation.

```
function DEFRAGMENT($taskset$, $schedule$, $time$)
    for all jobs $job$ in a window twice as large as the longest period in $taskset$ do
        $start\ index \leftarrow$ the index closest to $job$.release
        for all combinations of 2 executions, $e1$ and $e2$, of $job$ do
            unschedule $e1$ and $e2$
            if ! FORCESCHEDULE($schedule$, $job$, $e1$.execution + $e2$.execution, $start\ index$) then
                reschedule $e1$ and $e2$
            end if
        end for
    end for
end function
function FORCESCHEDULE($schedule$, $job$, $execution$, $index$)
    $reschedule\ index \leftarrow -1$
    while $index \leq schedule$.length and deadline at $index < job$.deadline do
        if can schedule $job$ at $index$ for $execution$ then
            return true
        else if can scheduled job at $index$ by shifting jobs then    ▷ An execution is allowed to be moved/exchanged with its neighbors if shifting it causes the job the be scheduled outside its window
            return true
        else if can schedule job at $index$ by rescheduling the left or/and the right job then
            $reschedule\ index \leftarrow index$
        end if
        $index \leftarrow index + 1$
    end while
    if $reschedule\ index \neq -1$ then
        schedule job at $rescedule\ index$ by moving the left or/and the right job outside its window
        return true
    end if
    return false
end function
```

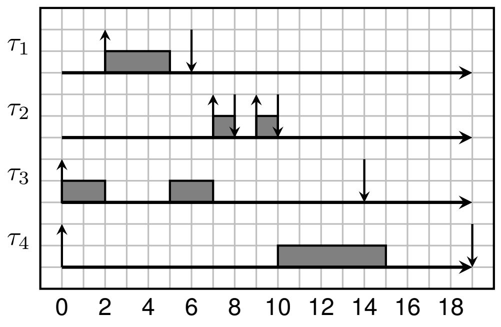


Figure 5.26: Before transformation 8-2 is applied

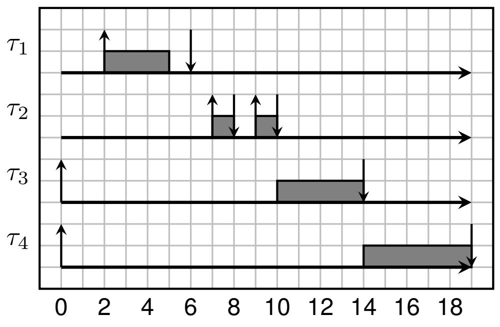


Figure 5.27: After transformation 8-2 is applied

5.3.4 Final Algorithm

The final algorithm is presented in Algorithm 5, where at each iteration:

1. if there are no active tasks, the schedule is de-fragmented using transformation 8.

2. if there are active tasks, and a job can be completed, one of transformations 6, 1, or 7 is invoked if possible, or the largest completing job is scheduled until its termination.

3. if there are active tasks, but no job can be finished, one of transformations 4, 3, 5, 6, or 2 is invoked if possible, otherwise the earliest deadline job is scheduled until the release of the higher priority job.

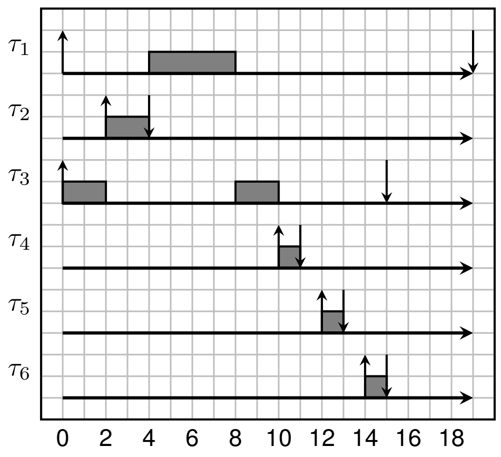


Figure 5.28: Before transformation 8-3 is applied

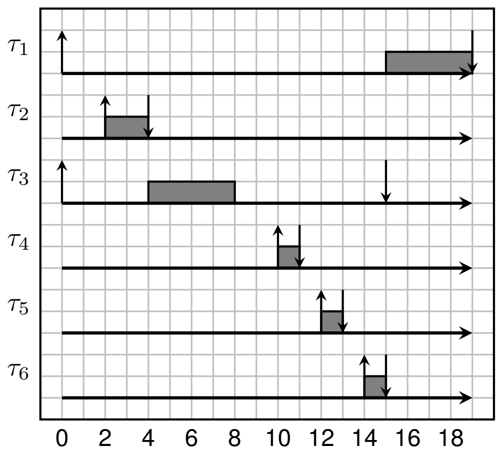


Figure 5.29: After transformation 8-3 is applied

The transformations have been ordered in least-cost greater benefit. Between equivalent actions, the application order does not seem relevant as each transformation treats a different scenario.

The algorithm presented here is the culmination of significant efforts towards an optimal scheduling algorithm. It is the first of its kind, as, as far as the authors of this thesis are aware, no other scheduling algorithm which follows a case-by-case scenario exists. Discovering these scenarios, and finding an efficient sub-algorithm to treat them is not an easy task. Alas, due to time constraints, it has not been possible to cover every existent scenario, particularly in the field of preemptive schedules. Nevertheless, the ones proposed here should be sufficient to evaluate the quality and validity of such a solution.

Algorithm 5 The final scheduling algorithm.

```
function SCHEDULE($taskset$)
    while schedule is not complete do
        if there are no active tasks then
            invoke transformation (8) Schedule Defragmentation
        else if can complete a job then
            if cannot apply transformation (6) Swap jobs when at least one can be scheduled non preemptively, (1) Relocate and Schedule Forward, or (7) Create the necessary space to remove a job out of a preemption zone by moving jobs which have a later deadline then
                complete the largest job possible
            end if
        else
            if cannot apply transformation (4) Complete a job by shrinking a preemption, (3) Extend the execution of a job , (5) Knapsack Jobs to prevent preemptions induced by very large tasks, (6) Swap jobs when at least one can be scheduled non preemptively, or (2) Reduce preemptions by splitting a job then
                schedule the earliest deadline job until the higher priority job is released
            end if
        end if
    end while
end function
```

5.4 Experimental Results

To evaluate our proposal, two *non-preemptive* methods in the state-of-the-art have been selected. Ideally, two preemptive equivalent competitors would have been chosen; unfortunately, no such solution exists that is not meta-heuristic/brute-force based.

The first method, BB-Moore [68], was developed by Moore. It is a branch-and-bound based pruning algorithm, where, any branch whose *tardiness*[2] *of all unscheduled jobs under EDF* is larger than any known branch is not explored.

The second method, Chain Window [69], was proposed by Nasri and Brandenburg and won an Outstanding Paper Award in RTAS 2017. Chain Window (CWin) is an iterative backtracking based algorithm that groups jobs in *chained windows*. Each chain window is comprised of a job sequence, a time window, and a slack value. Given a schedule produced by an online scheduling algorithm, such as NP-EDF or NP-RM, multiple chain windows are generated and merged as jobs which were previously deemed un-schedulable by the online algorithm are sequenced. By using slack to prune the research space, and further reducing this space by continuously merging each window, CWin can find a solution very efficiently, and represents the very best in schedulability and speed in the state-of-the-art.

To benchmark our proposal, dubbed PMin, against BB-Moore and Chain Window, we have obtained the task set used in [69]. This test-bed is composed of 1 000 task sets generated for a utilization level ranging from 10% to 100% in steps of 10%, totaling 9 000 sets. Each task set is limited to six tasks due to the fact resource constrained systems generally have a low number of tasks. For each task set, each scheduler has up to one minute to schedule the task; if a scheduler is unable to do so, the task set is reported as *undecided*.

The BB-Moore and Chain Window tests were conducted on a 3.0GHz Intel Xeon E7-8857 v2 with 16 cores, and 1.2 TiB of RAM. Our tests will be conducted on a much more humble 2011 Intel Core i7-2720QM 3.3GHz and 16 GB of RAM using scalameter – a microbenchmarking and performance regression testing framework for the JVM platform [91].

The results of this test are illustrated in figure 5.30 as a schedulability ratio, i.e., the percentage of tasks sets deemed schedulable in relation to the total number of generated task sets. In terms of schedulability, the algorithms exhibit similar results up to 70% utilization. From 70% on, CWin-EDF presents the highest schedulability ratio of any algorithm included in this analysis. CWin-EDF is trailed by BB-Moore's proposal by a 3% point difference. PMin and CWin-RM exhibit very similar performance throughout the range and offer the smallest schedulability ratio of the algorithms included here, with a difference of about 14% when compared to CWin-EDF.

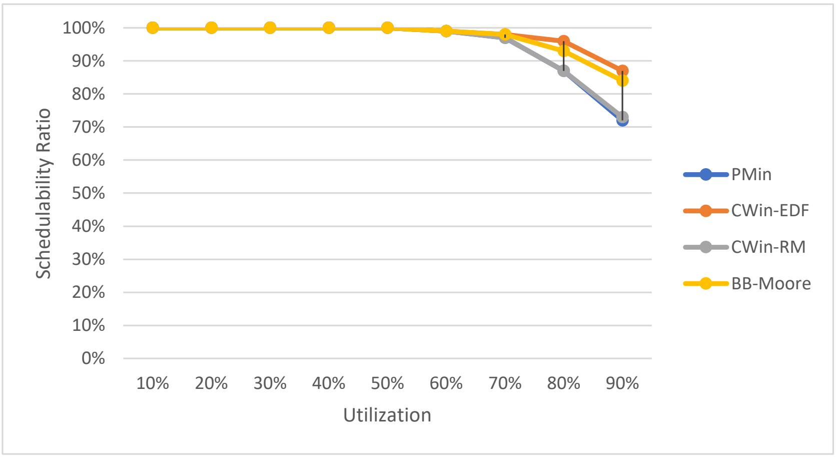


Figure 5.30: Schedulability ratio for each analyzed algorithm.

Given that PMin is a preemptive algorithm, it is possible to analyze how many jobs failed to be scheduled non-preemptively for each given task set. Figure 5.31, illustrates the ratio of non-preemptive jobs in task sets which could not be scheduled non-preemptively by PMin. The scheduler has a very high non-preemptive schedulability ratio, dropping to 99% at 60% utilization, and 98% at 90% utilization. More research is required to understand why PMin fails to join some jobs, and to adapt or develop more transformations if needed. Nevertheless, 98% of non-preemptive executions is a very good score for an algorithm which is initially targeting preemptive scheduling.

To obtain a measurable degree of certainty in the schedulability of our algorithm we have conducted a multitude statistical tests. Given that our data is nominal data, i.e., each task set is either *schedulable* or *unschedulable* non-preemptively, we have used the McNemar's statistical test to compare the three scheduling algorithms. Under a significance level of 5%, McNemar test

[2]Tardiness is defined as the job's deadline minus its completion time.

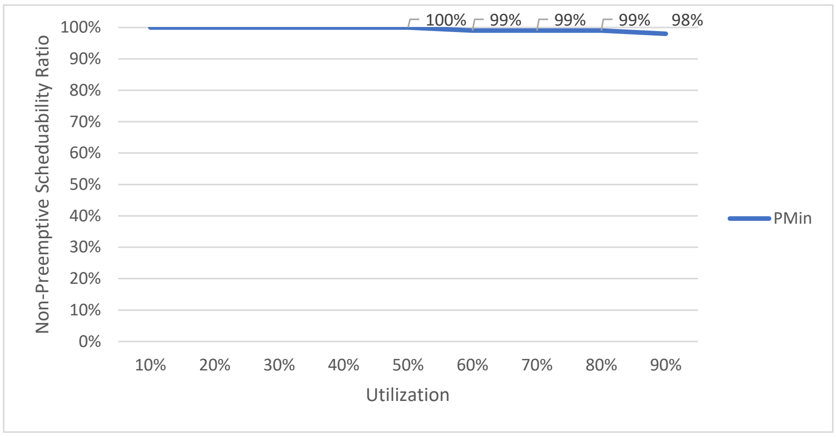


Figure 5.31: Non-Preemptiveness ratio for PMin.

fails to suggest there is a significant schedulability ratio difference between PMin and CWin-RM. However, between PMin and BB-Moore, the test indicates there is significant evidence to support the hypothesis that BB-Moore has a higher schedulability ratio than PMin. The full results of these tests can be found in appendix A.2 - Schedulability Analysis, on page 74.

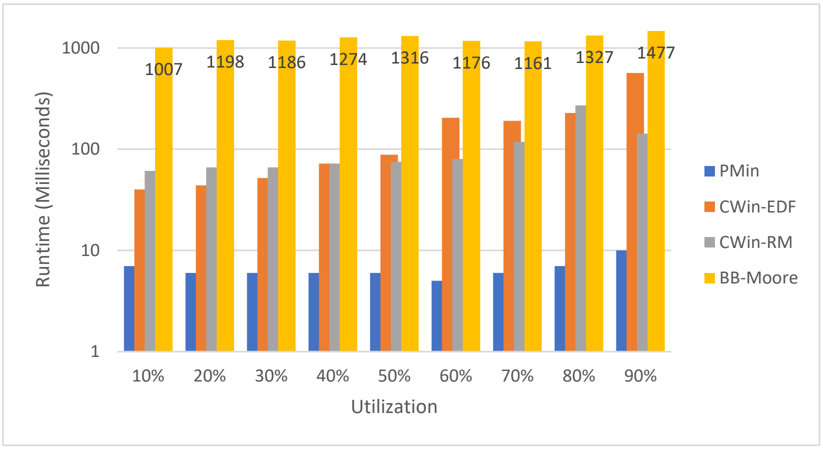


Figure 5.32: Average runtime for each scheduler when they successfully find a schedule. Note the logarithmic scale on the vertical axis.

Figure 5.32 contains average execution times for each utilization level. CWin-EDF, CWin-RM, and BB-Moore average runtimes were provided by Nasri and Brandenburg [69]. PMin values were extracted by scalabench; each task set was scheduled 36 times, a total of 36 000 runs for each utilization level.

Given the wide range of performance offered by each algorithm, the vertical axis of the plot in figure 5.32 is configured as a tenth base logarithm.

For what PMin consigns in schedulability, it quickly makes up in runtime, as it is the most efficient algorithm in this test by several orders of magnitude, even though it is a preemptive scheduling algorithm running on significantly slower hardware. Chain Window comes in at second place, followed by BB-Moore.

Statistical tests comparing PMin's and CWin-RM's performance are included in Appendix A.2 - Runtime Analysis, on page 76. Included in this section are a Shapiro-Wilk test, Anderson-Darling test, Lilliefors test, and a Jaque-Bera test, which provide a high degree of certainty PMin's and CWin-RM execution times on the test set are not normally distributed. Hence, we were only able to use the Sign Test and the Wilcoxon Signed-Rank test to compare the two algorithms. Both tests suggest there is sufficient evidence to assume PMin is, on average, faster than CWin-RM.

Result Analysis

Although PMin does not improve upon the current state-of-the-art in terms of schedulability, we must note that its competitors are designed exclusively for non-preemptive systems, and can therefor utilize superior techniques exclusive to the non-preemptive sub-problem. With this in mind, we believe the current solution to provide very good results indeed; and with more development time, there should be no reason why PMin's non-preemptive schedulability ratio cannot be increased by including more transformations. Surprisingly, PMin can provide a solution orders of magnitude faster than the current state-of-the-art solution, even while executing on slower hardware.

Chapter 6

Reference Implementation

Provides a detailed description of the algorithm's implementation and testing.

6.1 Architecture

Crucial to implementing a scheduler is its implementation environment. As far as the authors of this thesis are aware, no frameworks that provide the necessary environment for off-line scheduling that meet the requirements of our scheduler exist. Hence, we have developed our own from scratch.

In addition to the design principles described in the planning chapter, another key principle is the distinction between a definition of a scheduling algorithm and its implementation; in other words, the distinction between *what* and *how*. This leads to a multi-leveled architecture with two components: (1) schedulers – which define *what* has to be done to build a schedule, and (2) the framework – which contains the necessary instructions to perform the actions commanded by the schedulers – *how*.

Additionally, the modeling entities that define a job and an execution are *extensible*. A scheduler can easily embed additional information to a job, or the execution of a job, by creating their own subclasses of these entities. This is a much more elegant and efficient solution than storing this information in arrays or maps.

Finally, the scheduler makes a very high usage of *assertions* and *requirements* and can detect and report many bugs which would otherwise be very difficult to find. Each generated scheduled is guaranteed to be correct, i.e., all jobs and their computation requirements are scheduled within their execution windows.

The end result of such an architecture is an extremely flexible and bug-resistant framework that allows its users to easily define scheduling algorithms clearly and precisely, undisturbed by the implementation details such as slack computation, for example. Thanks to this, the main loop of the scheduler defined in this thesis looks much like the pseudo-code in Algorithm 5, defined the previous chapter.

Figure 6.1 contains a package diagram illustrating the multi-leveled architecture, where the builder package corresponds to the framework.

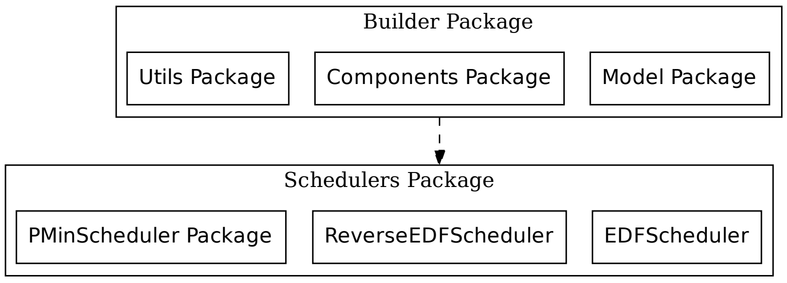


Figure 6.1: The main package diagram for the implementation.

Let's describe each entity in the package diagram:

Model Package: Contains the scheduling model as described in the system model.

Components Package: Houses all components directly related to generating a schedule, such as computing slack, applying transformations, etc.

Utils Package: Home to several support utilities that are not directly related to building a schedule, such as an algorithm to treat the knapsack problem.

PMinScheduler Package: Contains a reference implementation for the scheduling algorithm described in this thesis.

ReverseEDFScheduler: An implementation of the reverse earliest deadline scheduler as defined in section 5.2.1.

EDFScheduler: An implementation of the Earliest Deadline Scheduler.

6.1.1 Builder Package and Main Schedule Loop

The builder's *parent* component is the *ScheduleFactory* – a factory of schedules. The entity can be seen much like an implementation of the *template method* pattern as it is essentially a *schedule loop* that delegates scheduling decisions to a scheduler, and coordinates the necessary components – many of which are algorithms on their own – to support and implement the scheduler's decisions.

Because a *Scheduler* is a subtype of IScheduler, it is a scheduler and not a schedule factory, hence the implementation deviates from the standard template method pattern in the sense that a scheduler does not extend the ScheduleFactory but rather provides the necessary methods as high order functions during the instantiation of the ScheduleFactory.

The schedule and the factory communicate via two objects, a *ScheduleState* and a *ScheduleDecision.* The *ScheduleState* is a "gateway" to the schedule through which several operations can be performed, the most basic of which include: generating a list of all possible job executions, computing the max-runtime for each execution, and extracting the maximum idle-time at time T; the more complex operations involve schedule rewrites, and are performed by a dedicated component known as *ScheduleRewriter*, also accessible via the *ScheduleState.* The *ScheduleDecision* is responsible for accepting the scheduler's decision and coordinating its implementation. The *ScheduleState* bounds all valid decisions explicitly, and if an invalid decision or no decision at all is provided, it is assumed to be an unrecoverable error and an exception is thrown.

A state diagram describing the main schedule loop is provided in figure 6.2. When a schedule is instantiated, a schedule factory is instantiated with two higher-order functions as parameters: the accept function which binds the scheduler to the ScheduleDecision and ScheduleState objects, and the schedule method which makes a scheduling decision at each iteration. When the generate method is invoked, the factory instantiates all necessary objects, such as but not limited to: the *ScheduleDecision*, the *ScheduleState*, the *ScheduleBuilder* – a component which appends jobs to the schedule as it is built, and the ScheduledJobTree – the entity that tracks active/pending jobs and maintains their slack. During each iteration, the schedule state produces a table with all possible executions and their maximum execution times, the maximum idle period, and upcoming releases; the scheduler then makes a decision upon this information. Once all jobs have been scheduled, the factory invokes the *build schedule* method on the ScheduleBuilder, which produces the final *Schedule* object.

Figure 6.3 contains a class diagram for all classes within builder, where:

ScheduleFactoryConfig This object provides configuration parameters to the ScheduleFactory, currently two options are supported: (1) hideIdleTimes which skips periods where no active job exists, and (2) slackCalculator, which allows the scheduler to choose one of the three slack calculating algorithms defined in the previous chapter.

ScheduleFactory The ScheduleFactory is instantiated by a Scheduler; as a generic class, it requires a Job and ScheduledJob types, which allows the Scheduler to append information to a Job or ScheduledJob. In addition, it also requires: a job factory, which can instantiate jobs of type J; a scheduled job factory that instantiates scheduled jobs of type JS; an optional scheduledJobReleasedListener, if the scheduler wishes to be notified when a job is released; an accept method, which configures the scheduler to use the correct ScheduleDecision and ScheduleState objects; and finally a schedule method, which invokes the scheduler at each iteration.

The ScheduleFactory makes use of the following components:

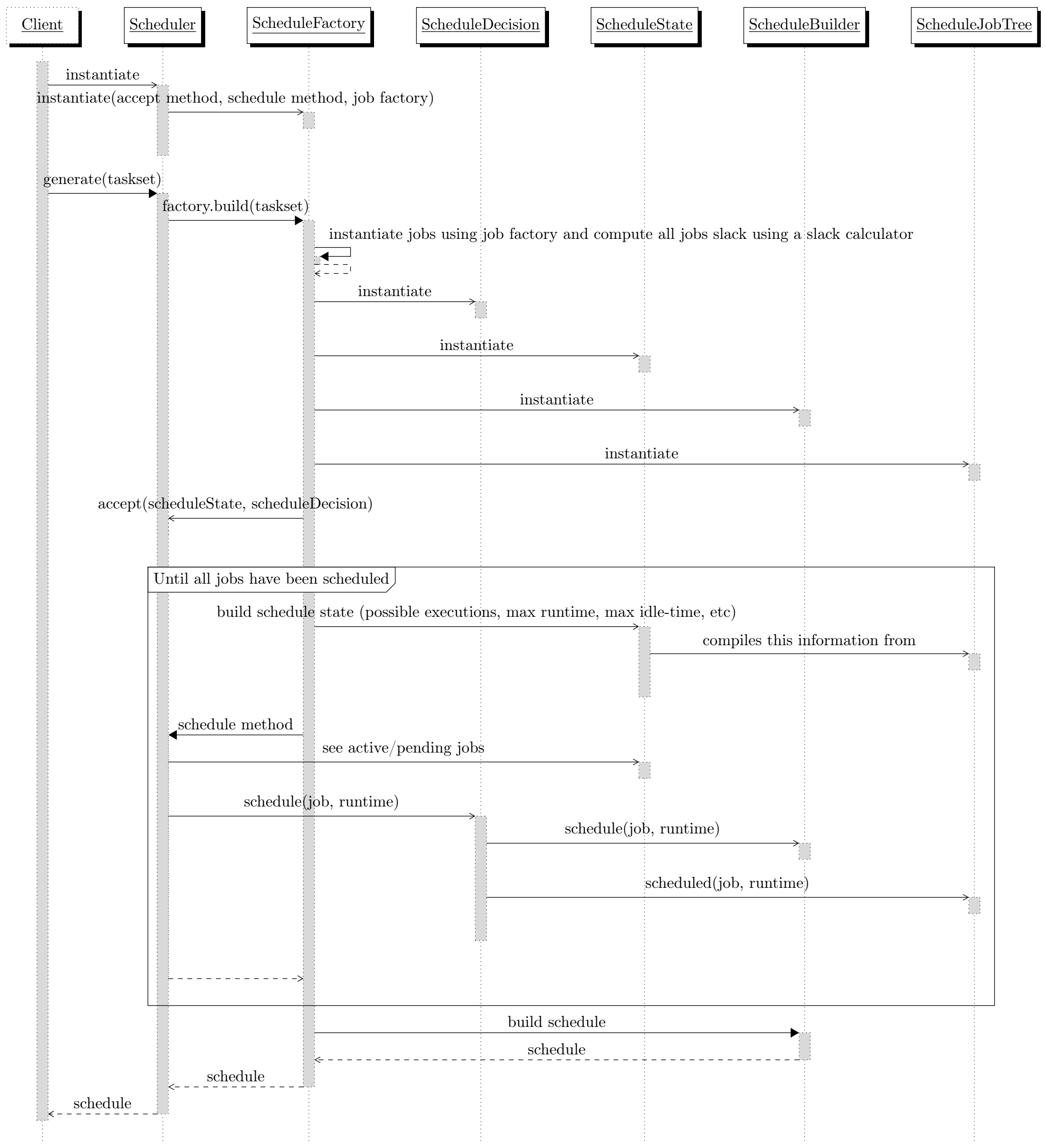


Figure 6.2: A sequence diagram for the main schedule loop.

ScheduleJobTree Tracks active/pending jobs and maintains job's slack at runtime.

ScheduleState Provides an overview of all possible execution options, along with their maximum execution times and the respective idle period. In addition, it also provides access to all remaining public components, namely: the ScheduleRewriter, and the ScheduleBuilder.

ScheduleDecision The object which is responsible for coordinating a decision made by the scheduler. All decisions are required to be valid otherwise an exception is thrown.

ScheduleRewriter Gateway to all rewriting operations.

ScheduleBuilder Responsible for adding/removing jobs from the schedule, safe iteration of such schedule, and finally once the schedule is built, generating the Schedule object. The ScheduleBuilder also provides several transformations that may be applied to the schedule.

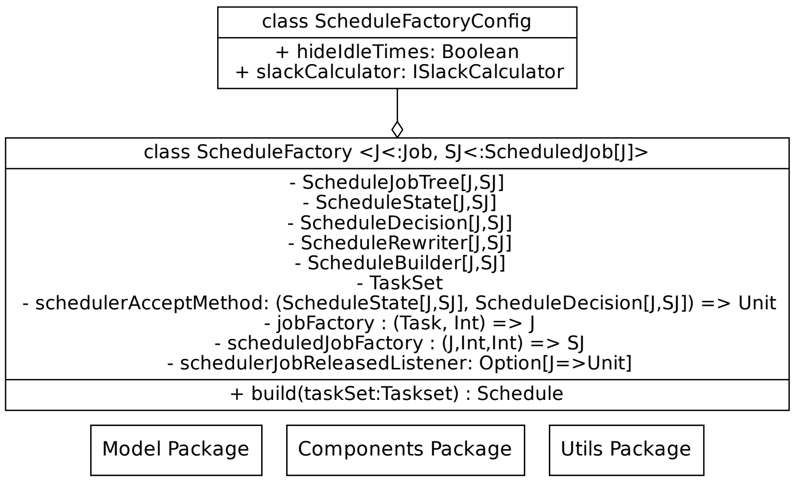

class ScheduleFactoryConfig
+ hideIdleTimes: Boolean
+ slackCalculator: ISlackCalculator
class ScheduleFactory <J<:Job, SJ<:ScheduledJob[J]>
- ScheduleJobTree[J,SJ]
- ScheduleState[J,SJ]
- ScheduleDecision[J,SJ]
- ScheduleRewriter[J,SJ]
- ScheduleBuilder[J,SJ]
- TaskSet
- schedulerAcceptMethod: (ScheduleState[J,SJ], ScheduleDecision[J,SJ]) => Unit
- jobFactory : (Task, Int) => J
- scheduledJobFactory : (J,Int,Int) => SJ
- schedulerJobReleasedListener: Option[J=>Unit]
+ build(taskSet:Taskset) : Schedule
Model Package
Components Package
Utils Package

Figure 6.3: A class diagram for all classes within Builder.

6.2 Package description

6.2.1 Model Package

The model package contains an object mapping of the system model described in section 5.1, although given the framework's complexity, it is unsurprising that a few adaptations have been made. A class diagram for the model package is presented in figure 6.4, where:

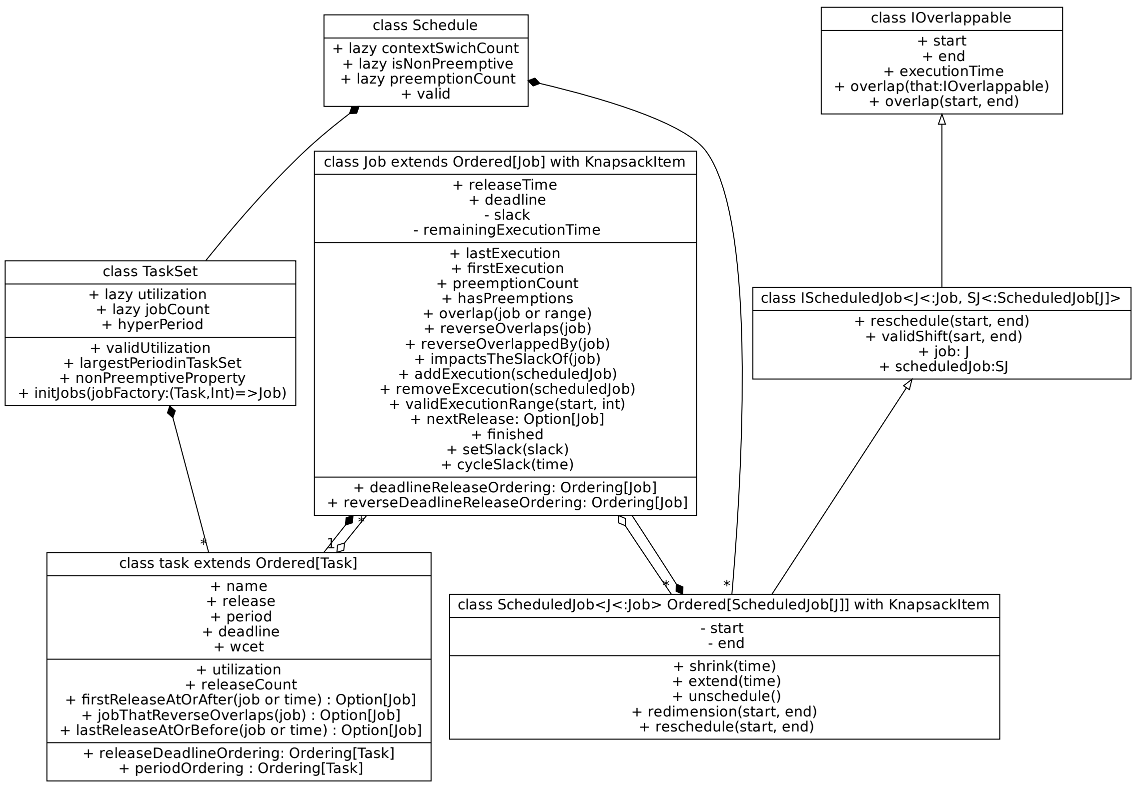

class Schedule
+ lazy contextSwichCount
+ lazy isNonPreemptive
+ lazy preemptionCount
+ valid
class IOverlappable
+ start
+ end
+ executionTime
+ overlap(that:IOverlappable)
+ overlap(start, end)
class Job extends Ordered[Job] with KnapsackItem
+ releaseTime
+ deadline
- slack
- remainingExecutionTime
+ lastExecution
+ firstExecution
+ preemptionCount
+ hasPreemptions
+ overlap(job or range)
+ reverseOverlaps(job)
+ reverseOverlappedBy(job)
+ impactsTheSlackOf(job)
+ addExecution(scheduledJob)
+ removeExcecution(scheduledJob)
+ validExecutionRange(start, int)
+ nextRelease: Option[Job]
+ finished
+ setSlack(slack)
+ cycleSlack(time)
+ deadlineReleaseOrdering: Ordering[Job]
+ reverseDeadlineReleaseOrdering: Ordering[Job]
class TaskSet
+ lazy utilization
+ lazy jobCount
+ hyperPeriod
+ validUtilization
+ largestPeriodinTaskSet
+ nonPreemptiveProperty
+ initJobs(jobFactory:(Task,Int)=>Job)
class IScheduledJob<J<:Job, SJ<:ScheduledJob[J]>
+ reschedule(start, end)
+ validShift(sart, end)
+ job: J
+ scheduledJob:SJ
*
1
*
*
*
class task extends Ordered[Task]
+ name
+ release
+ period
+ deadline
+ wcet
+ utilization
+ releaseCount
+ firstReleaseAtOrAfter(job or time) : Option[Job]
+ jobThatReverseOverlaps(job) : Option[Job]
+ lastReleaseAtOrBefore(job or time) : Option[Job]
+ releaseDeadlineOrdering: Ordering[Task]
+ periodOrdering : Ordering[Task]
class ScheduledJob<J<:Job> Ordered[ScheduledJob[J]] with KnapsackItem
- start
- end
+ shrink(time)
+ extend(time)
+ unschedule()
+ redimension(start, end)
+ reschedule(start, end)

Figure 6.4: A class diagram for all classes within Model.

Schedule defines a *valid* schedule produced by a scheduler.

TaskSet a class that aggregates multiple tasks into a task set.

Task represents a system task.

Job is a job of a task.

ScheduledJob is an execution of a Job.

IScheduledJob abstracts an execution of a job. This class enables the scheduler to simulate/apply a transformation without knowing if it is working on dummy objects or the actual executions.

IOverlappable defines a type of object may overlap with another of its type in a dimension with one vector.

Class Job and ScheduledJob inherit the KnapsackItem trait which enables them to be used with the Knapsack algorithm defined in the util package. Most class methods should be implicit; hence we will only describe those whose usage may be ambiguous:

TaskSet: nonPreemptivePropery This method checks if the schedule possesses a *required* condition which any non-preemptive task set must comply with, but is not *sufficient* to prove a taskset can be scheduled non preemptively. The property checks if all existing tasks can be scheduled between two releases of the system tasks with the shortest period, such that the computational and timing requirements of these tasks are respected.

Task: jobThatReverseOverlaps(job):Job returns a job of this task that carries out the job provided in the parameter.

Job: reverseOverlaps(job) if the job carries out the job specified as a parameter.

reverseOverlappedBy(job) if the job is carried out by the job specified as a parameter.

ScheduledJob redimension(start, end) redimensions this execution to begin and start at end, the redimensioning must not extend beyond other executions of the same job.

reschedule(start, end) reschedules the execution while at the same time updating the ordered list of executions of the job this execution belongs to.

IScheduledJob validShift(start, end) returns true if the start and end is a period within the execution window of its parenting job.

6.2.2 Components Package

The components package contains all components directly related to generating a schedule, such as computing slack, applying transformations, etc. A class diagram illustrating its contents is displayed in figure 6.5, where:

ScheduleDecision Is responsible for coordinating the scheduler's decision at each iteration.

ScheduleState Provides an overview of all possible execution options, along with their maximum execution times and the maximum idle period at time T. In addition, it also provides access to all the scheduler's remaining public components, such as the ScheduleBuilder, and the ScheduleRewriter. The ScheduleState is composed of a static Array of T ScheduleOptions, where T is the number system tasks.

ScheduleOption Defines a possible scheduling decision of a job at a given iteration, while bounding its maximum execution time.

ScheduleBuilder Responsible for adding/removing jobs from the schedule, safe iteration of said schedule, and finally once the schedule is built, generating the Schedule object. The ScheduleBuilder also provides several transformations that may be applied to the schedule.

ScheduleJobTree Tracks active/pending jobs and maintains job's slack at runtime.

ScheduleRewriter Gateway to all rewriting operations.

The ScheduleDecision provides four decision types:

schedule which schedules a schedule option. The decision makes sure the runtime value provided by the scheduler does not exceed the bound defined in the schedule option. If no duration is specified, the decision schedules the option for as long as possible.

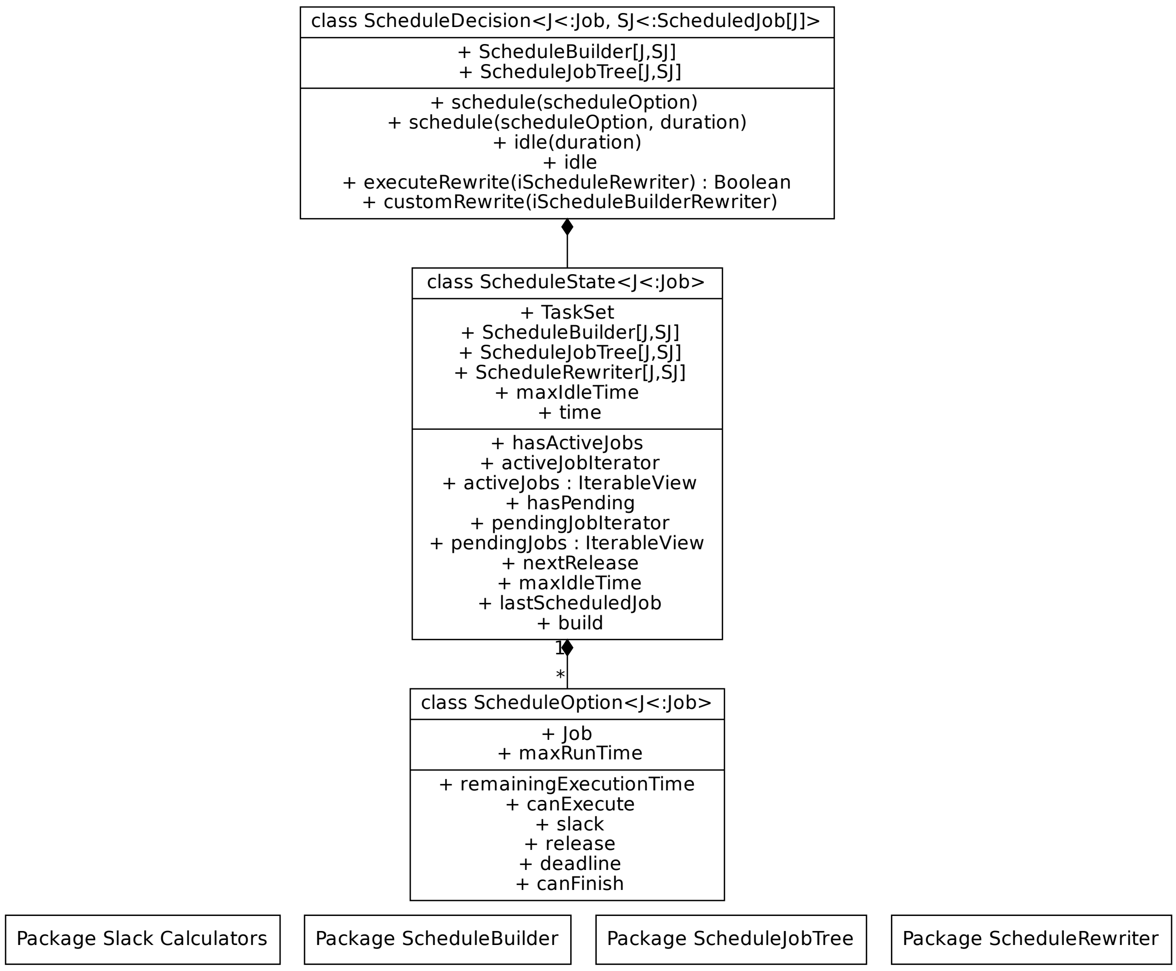


Figure 6.5: A class diagram for all classes within the components package.

idle which schedules an idle period. The decision ensures the period duration is below the bound defined in the ScheduleState. If no duration is specified, the decision schedules the period until a job is released.

rewrite attempts to schedule a standard rewrite performed by the ScheduleRewriter Components.

customRewrite applies a custom rewrite to the scheduler that is applied by an IScheduleBuilder-Rewriter – a component responsible for the safe iteration of a Schedule.

The ScheduleState's build method is invoked by the ScheduleFactory on each iteration to build the ScheduleOption table, which contains all possible jobs which can be scheduled at that iteration, along with their maximum run-time, and the maximum idle time permitted within that iteration. This information is presented by the State but is compiled by the ScheduleJobTree.

6.2.3 Slack Calculator Package

The slack calculator package contains an implementation of all slack computation methods described in section 5.2.

The class hierarchy follows the Strategy pattern, where ISlackCalculator defines a family of algorithms to compute slack, and the remaining classes are implementations of these algorithms. AlgorithmicSlackCalculator, MathSlackCalculator, and ReverseEDFSlackCalculator correspond to the schedulers defined in 5.2, where the AlgorithmicSlackCalculator extracts the slack by simulating a rEDF scheduler, and the ReverseEDFSlackCalculator extracts the slack from a schedule produced by a rEDF scheduler.

The ISlackCalculator class also contains a method called verifySlackValues which compares the slack of all jobs between two equal task sets.

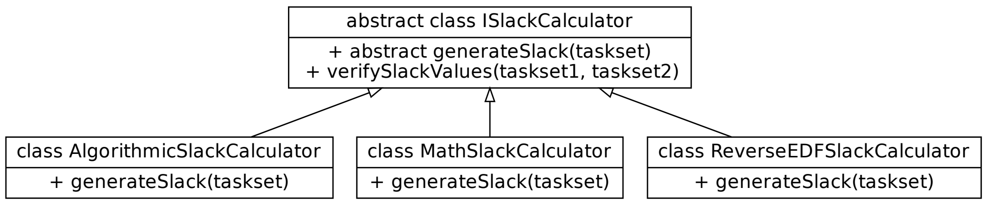

abstract class ISlackCalculator
+ abstract generateSlack(taskset)
+ verifySlackValues(taskset1, taskset2)
class AlgorithmicSlackCalculator
+ generateSlack(taskset)
class MathSlackCalculator
+ generateSlack(taskset)
class ReverseEDFSlackCalculator
+ generateSlack(taskset)

Figure 6.6: A class diagram for all classes within the components package.

6.2.4 ScheduleJobTree Package

The *ScheduleJobTree* is the framework's data structure responsible for queuing and dequeuing system jobs, as well as maintaining their slack. It is comprised of two sorted sets, one managing active jobs, and another administering pending releases; at any time T, if a task τ_i has uncompleted jobs, a single job of τ_i must be present in either the active queue or the pending queue. At runtime, when a job J_i is executed, its remaining execution time is updated by the ScheduleBuilder; if the remaining time is zero, the job is complete, and the ScheduleJobTree will remove J_i from the active queue and allocate its next release in the active or pending queue.

The slack computation is deeply related with the two queue mechanism of the data structure. The active queue is sorted in earliest deadline first, while the pending queue is governed in earliest release first. This priority order enables efficient processing and management of slack, since:

1. Extracting the maximum runtime of a job requires iterating all lower priority jobs plus any job released within that period.
2. Updating the slack of all jobs after scheduling a job requires iterating all active tasks, plus any pending job that is released within that period – which will have its slack reduce in equal amount to the overlap between the execution of the active and pending job.
3. Scheduling an idle period requires reducing the slack of any job that is active within that period.

Iterating all active jobs of lower priority than J_i is simple since those jobs appear first in the sorted set. Iterating all pending jobs that overlap with the current execution is also easily bounded, since, given the earliest release priority order, once a job J_i no longer overlaps with the scheduled execution, no other jobs in the set can, as their release time are increasing and can only be larger than J_i's.

Rewrite operations are supported through a dedicated component known as *ScheduleJob-TreeRewriter*. The Rewriter, instantiated by the ScheduleJobTree, clones the active and pending queues wrapping any job in either queue with a dummy object which will temporarily hold any changes to a job's slack or remaining execution time. Although tracking the remaining execution time of a job is not the responsibility of the ScheduleJobTree, the rewriter component nevertheless needs to know if the job is complete as reschedule operations are applied. In addition to the active and pending queues, a new queue contains all new active jobs which are released due to schedule operations, or previously completed jobs which are moved to the active state by being unscheduled. Once the rewrite operation is verified to be valid, and the *apply* method is invoked exclusively by the ScheduleDecision, the rewriter will merge with its parenting ScheduleJobTree, modifying job's slack directly, but updating the Tree's queues by invoking package-private methods, effectively marking all operations as permanent.

One limitation of the rewriter is that it does not enqueue any new pending jobs past the previous pending job of any task; this occurs exclusively during rollback operations, and currently, no transformation in the schedule performs exclusive rollback operations which cause their previous release to be moved to the pending state.

A class diagram is provided in figure 6.7. Most class methods should be implicit; hence, we will only detail those that are hard to understand:

Most methods in the class diagram should be implicit; hence we will only describe those who are more difficult to understand. In the ScheduleJobTree, the scheduled and idle methods are invoked exclusively by the ScheduleDecision under normal schedule or idle operations. The package-private releaseJob, completeJob, and uncompleteJob are operated by the ScheduleJobTreeRewriter

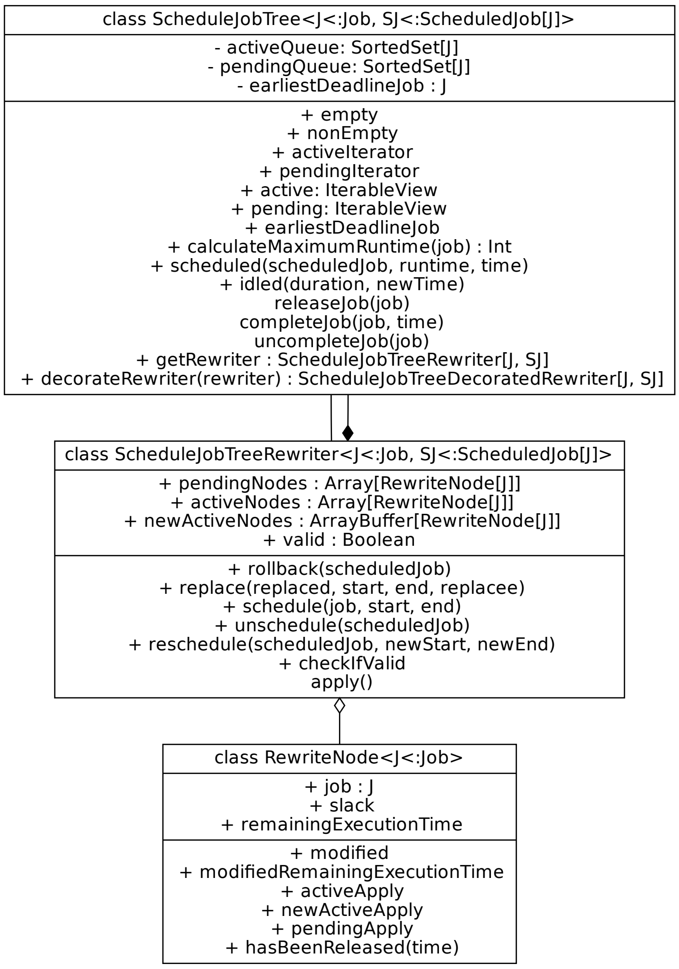

class ScheduleJobTree<J<:Job, SJ<:ScheduledJob[J]>
- activeQueue: SortedSet[J]
- pendingQueue: SortedSet[J]
- earliestDeadlineJob : J
+ empty
+ nonEmpty
+ activeIterator
+ pendingIterator
+ active: IterableView
+ pending: IterableView
+ earliestDeadlineJob
+ calculateMaximumRuntime(job) : Int
+ scheduled(scheduledJob, runtime, time)
+ idled(duration, newTime)
releaseJob(job)
completeJob(job, time)
uncompleteJob(job)
+ getRewriter : ScheduleJobTreeRewriter[J, SJ]
+ decorateRewriter(rewriter) : ScheduleJobTreeDecoratedRewriter[J, SJ]
class ScheduleJobTreeRewriter<J<:Job, SJ<:ScheduledJob[J]>
+ pendingNodes : Array[RewriteNode[J]]
+ activeNodes : Array[RewriteNode[J]]
+ newActiveNodes : ArrayBuffer[RewriteNode[J]]
+ valid : Boolean
+ rollback(scheduledJob)
+ replace(replaced, start, end, replacee)
+ schedule(job, start, end)
+ unschedule(scheduledJob)
+ reschedule(scheduledJob, newStart, newEnd)
+ checkIfValid
apply()
class RewriteNode<J<:Job>
+ job : J
+ slack
+ remainingExecutionTime
+ modified
+ modifiedRemainingExecutionTime
+ activeApply
+ newActiveApply
+ pendingApply
+ hasBeenReleased(time)

Figure 6.7: A class diagram for all classes within the ScheduleJobTree package.

whenever a rewrite occurs, and the ScheduleJobTree's queues must be maintained by shifting jobs between the active and pending queues.

Finally, the RewriteNode methods, where modified returns true if the slack or remaining execution times were modified in any way, and the remaining apply methods are invoked during rewrite operations depending on the queue the RewriteNode is located, i.e., if it is located in the active queue, the activeApply method will be invoked.

6.2.5 ScheduleBuilder Package

The ScheduleBuilder is the authority responsible for building, iterating, and applying very well defined transformations to a schedule. Given these responsibilities, it is the only entity in the entire scheduling framework with the right and the means to instantiate a ScheduledJob. This ensures the execution list of each job is maintained correctly, where all executions of a job are properly ordered within this list.

The ScheduleBuilder also provides the necessary constructs to iterate and modify the schedule, known as *ScheduleBuilderRewriters*. There are many types of rewriters, each crafted to a specific use case:

RawRewriter: The Raw rewriter operates directly on the schedule, hence all operations are immediately applied and viewable when queried.

CloningRewriter: A Cloning rewriter forks a portion of a schedule, wrapping executions with a wrapper object, and applies several transformations to this copy. Once the apply method is invoked, the CloningRewriter merges the copy with its original. Any operation to the rewriter is immediately visible.

CachedRewriter: The Cached Rewriter supports only the schedule operation. It is useful when Knapsacking multiple jobs within a given period as their release-deadline bounds may prevent the job set from being scheduled. No view operation is supported.

DummyRewriter: The Dummy Rewriter supports all operations but silently ignores any modification request and is useful to test if the transformation is coherent a slack perspective. Not all transformations can be simulated with a dummy rewriter, but most are written such that they do not perform unnecessary gets.

Of all the re-writers, only the Cloning Rewriter is partially implemented; not only because it is memory intensive but primarily because it requires expanding the schedule clone when a transformation exceeds past its bounds. The additional complexity, given the necessity for a mechanism to compensate the fact that all indexes prior to the expansion are invalid, ultimately made the approach too costly. Since the rewriters share a common interface know as the IScheduleBuilderRewriter, the scheduler does not need to know what implementation of rewriter it is working with.

In addition, each rewriter can be *decorated* with two entities:

BackTrackingRewriter: The BacktrackingRewriter permits hierarchical commit and rollback operations. Hence, a scheduler can at any point in time save the current schedule state on any Rewriter, and undo all operations until the last save by invoking rollback. Commit and rollback operations are hierarchical, and nested saves and rollbacks are supported. The BackTrackingRewriter operates by saving each move performed by a scheduler into an array. When the save method is invoked, the Rewriter saves the active array and creates a one. When rollback is invoked, the Rewriter rolls back all actions in the current array, and sets the previous one as active.

ScheduleJobTreeDecoratedRewriter: The JobTreeDecoratedRewriter wraps any rewriter such that any modification is tracked by a ScheduleJobTreeRewriter. During the transformation, the scheduler can query the rewriter if the slack tables represent a valid schedule. When the transformation is finished and the apply method is invoked, the ScheduleJobTreeDecoratedRewriter will update its parenting ScheduleJobTree's slack table, ensuring coherency with the produced schedule. Like the BackTrackingRewriter, this rewriter also supports hierarchical commit and rollback operations, where at each commit a copy of the ScheduleJobTreeRewriter is saved, and on each rollback, a copy is restored.

A class diagram containing all rewriters is provided in figure 6.8. The IScheduleBuilderRewriter contains abstract methods defining each schedule operation, namely *get, schedule, rollback, unschedule, reschedule* and *re-dimension.* The remaining operations are facilitating methods when the user does not know at which index to schedule an execution where the Rewriter trait will search for the correct insertion index with *find*, invoke the corresponding abstract operation.

The valid method returns true if the schedule is valid; when the rewriter is decorated with a slack aware rewriter, then valid's return code will also depend on the validity of the system slack. The find method will search for the closest index using *binary search.* The mergeCommit merges the two last commits into one. Finally, the utils method returns a ScheduleBuilderUtils type whose only function implements the forceSchedule algorithm defined in the schedule deframenter algorithm 4, on section 5.3.3.

Missing from the diagram are the supporting IBacktrackSavedMove classes, which support the BackTrackingRewriter. These classes are depicted in the class diagram on figure 6.9, where each subclass of IBacktrackSavedMove contains an algorithm to undo a *schedule, unschedule, redimension,* or *reschedule* move.

The final subcomponent of IScheduleBuilder is responsible for enacting simple transformations such as de-scheduling or extending jobs. Each transformation implements the IScheduleTransformer trait, depicted in figure 6.10, which defines a single function that applies a transformation through a rewriter, and returns true if the transformation was successful.

The following transformers have been defined:

ChainTransformer: The chain transformer combines multiple transformations by invoking them sequentially. A class diagram depicts the class in figure 6.11. The transformation is said to be successful if all its sub-transformations were applied correctly.

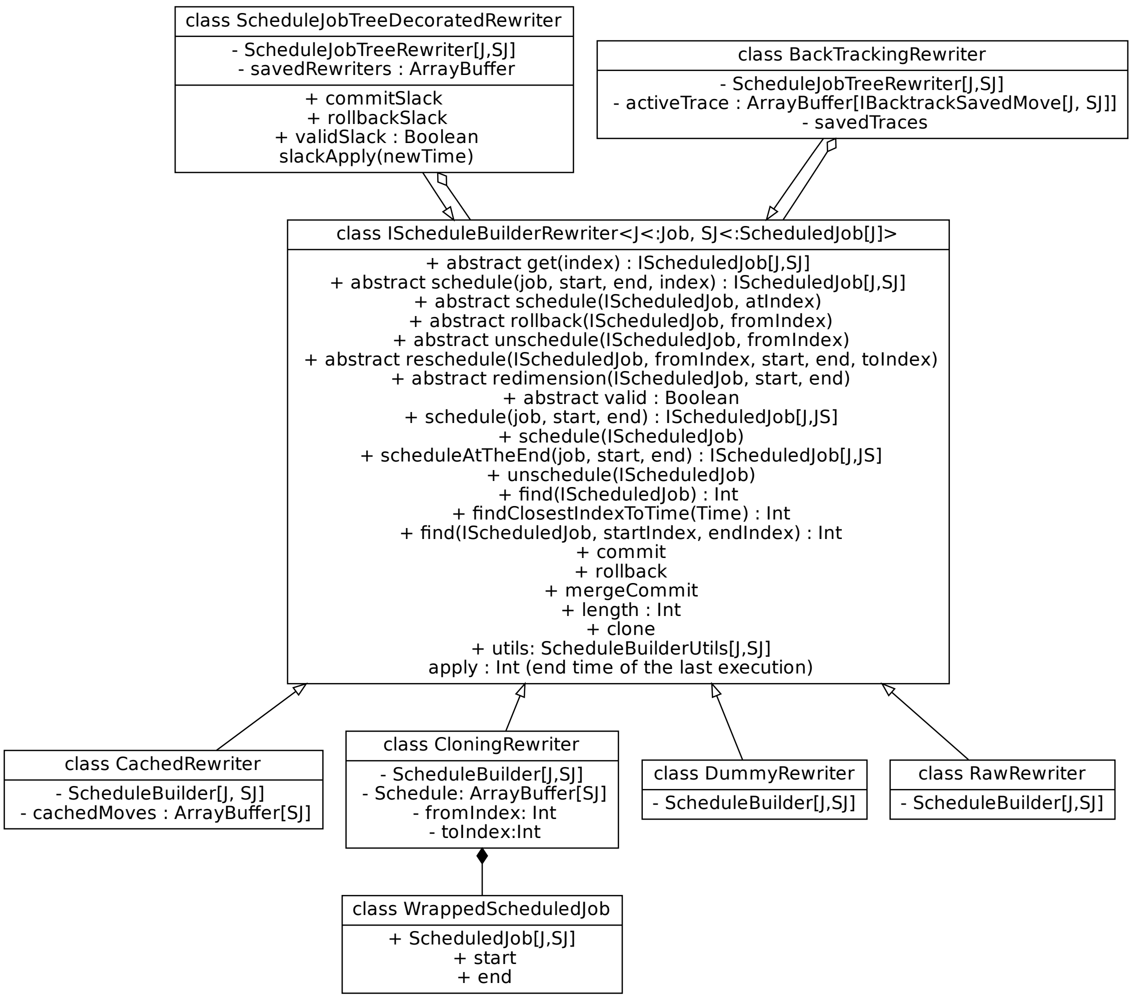

class ScheduleJobTreeDecoratedRewriter
- ScheduleJobTreeRewriter[J,SJ]
- savedRewriters : ArrayBuffer
+ commitSlack
+ rollbackSlack
+ validSlack : Boolean
slackApply(newTime)
class BackTrackingRewriter
- ScheduleJobTreeRewriter[J,SJ]
- activeTrace : ArrayBuffer[IBacktrackSavedMove[J, SJ]]
- savedTraces
class IScheduleBuilderRewriter<J<:Job, SJ<:ScheduledJob[J]>
+ abstract get(index) : IScheduledJob[J,SJ]
+ abstract schedule(job, start, end, index) : IScheduledJob[J,SJ]
+ abstract schedule(IScheduledJob, atIndex)
+ abstract rollback(IScheduledJob, fromIndex)
+ abstract unschedule(IScheduledJob, fromIndex)
+ abstract reschedule(IScheduledJob, fromIndex, start, end, toIndex)
+ abstract redimension(IScheduledJob, start, end)
+ abstract valid : Boolean
+ schedule(job, start, end) : IScheduledJob[J,JS]
+ schedule(IScheduledJob)
+ scheduleAtTheEnd(job, start, end) : IScheduledJob[J,JS]
+ unschedule(IScheduledJob)
+ find(IScheduledJob) : Int
+ findClosestIndexToTime(Time) : Int
+ find(IScheduledJob, startIndex, endIndex) : Int
+ commit
+ rollback
+ mergeCommit
+ length : Int
+ clone
+ utils: ScheduleBuilderUtils[J,SJ]
apply : Int (end time of the last execution)
class CachedRewriter
- ScheduleBuilder[J, SJ]
- cachedMoves : ArrayBuffer[SJ]
class CloningRewriter
- ScheduleBuilder[J,SJ]
- Schedule: ArrayBuffer[SJ]
- fromIndex: Int
- toIndex:Int
class DummyRewriter
- ScheduleBuilder[J,SJ]
class RawRewriter
- ScheduleBuilder[J,SJ]
class WrappedScheduledJob
+ ScheduledJob[J,SJ]
+ start
+ end

Figure 6.8: A class diagram describing the ScheduleBuilder Rewriters.

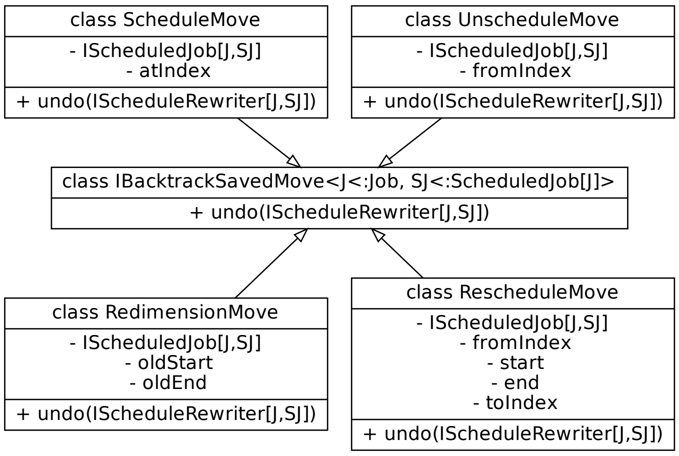


Figure 6.9: A class diagram describing the Backtacking moves hierarchy.

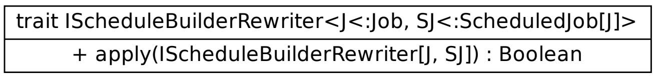
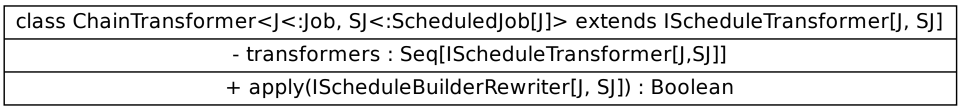

trait IScheduleBuilderRewriter<J<:Job, SJ<:ScheduledJob[J]>
+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.10: A class diagram describing the IScheduleBuilderRewriter trait.

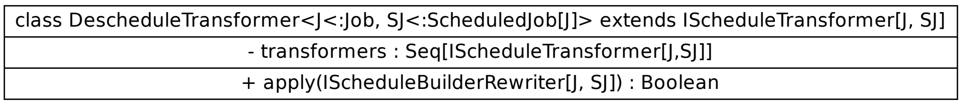


class ChainTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]

- transformers : Seq[IScheduleTransformer[J,SJ]]

+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.11: A class diagram describing the ChainTransformer class.



class DescheduleTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]

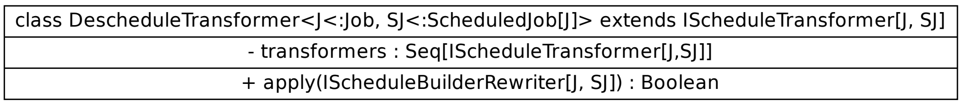
- transformers : Seq[IScheduleTransformer[J,SJ]]

+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.12: A class diagram describing the DescheduleTransformer class.

DescheduleTransformer De-schedules a sequence of ScheduledJobs from the Schedule. The class is depicted in figure 6.12.

ExtendTransformer Extends the execution of a task, shifting other tasks to the right. The class is depicted in figure 6.13.



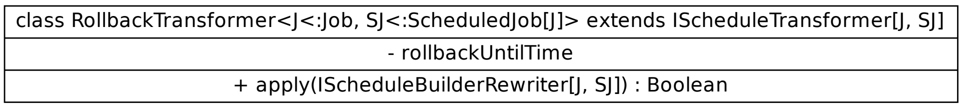
class DescheduleTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]

- transformers : Seq[IScheduleTransformer[J,SJ]]

+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.13: A class diagram describing the ExtendTransformer class.

RollbackTransformer Rolls back time to a predefined value, in other words, un-schedules all jobs until time T. A class diagram describing this transformation is included in figure 6.14



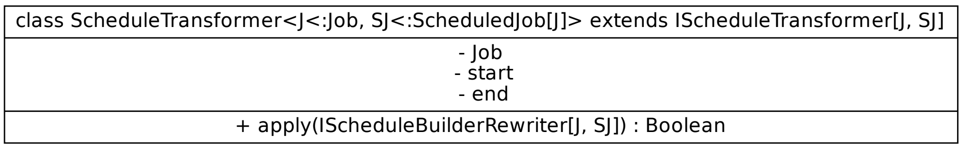
class RollbackTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]

- rollbackUntilTime

+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.14: A class diagram describing the RollbackTransformer class.

ScheduleTransformer Schedules a job in the schedule and is illustrated in figure 6.15.



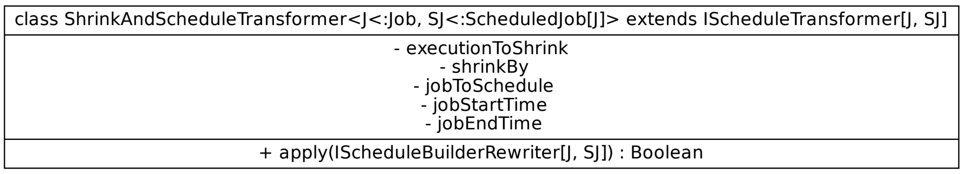
class ScheduleTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]

- Job
- start
- end

+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.15: A class diagram describing the RollbackTransformer class.

ShrinkAndScheduleTransformer shrinks an execution and pushes jobs backward, creating enough space to schedule a job non-preemptively. Executions may have to be switched if they breach their execution window. Class is depicted in figure 6.16.



class ShrinkAndScheduleTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]

- executionToShrink
- shrinkBy
- jobToSchedule
- jobStartTime
- jobEndTime

+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.16: A class diagram describing the RollbackTransformer class.

SplitAndScheduleTransformer Splits an execution, leaving one part in place and scheduling the other part in the past, pushing jobs backward and creating enough space to schedule a job non-preemptively. The class is illustrated in figure 6.17.

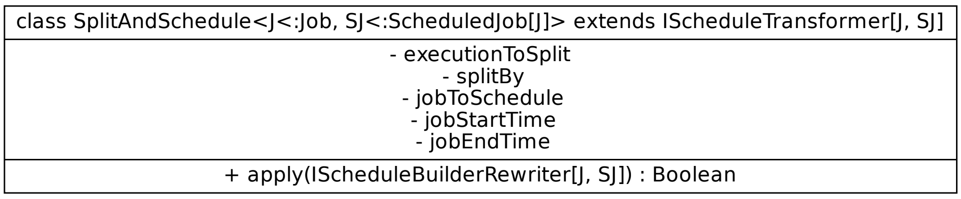

class SplitAndSchedule<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]
- executionToSplit
- splitBy
- jobToSchedule
- jobStartTime
- jobEndTime
+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.17: A class diagram describing the RollbackTransformer class.

SwitchAndExtendTransformer Switches the execution of two executions, extending one of them. A class diagram depicting the class is included in figure 6.18.

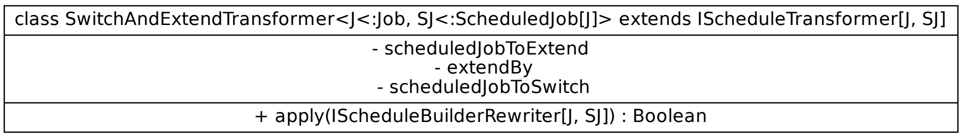

class SwitchAndExtendTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]
- scheduledJobToExtend
- extendBy
- scheduledJobToSwitch
+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.18: A class diagram describing the SwitchAndExtendTransformer class.

SwitchTransformer Switches multiple executions of a job J_i by a job J_j, the executions are combined if any execution J_i neighbors J_j. The class is depicted in figure 6.19.

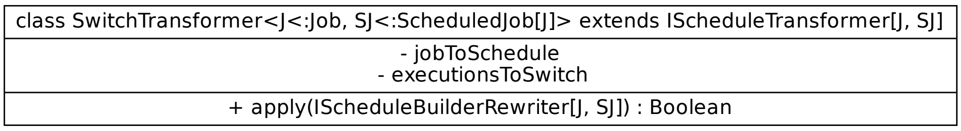

class SwitchTransformer<J<:Job, SJ<:ScheduledJob[J]> extends IScheduleTransformer[J, SJ]
- jobToSchedule
- executionsToSwitch
+ apply(IScheduleBuilderRewriter[J, SJ]) : Boolean

Figure 6.19: A class diagram describing the SwitchTransformer class.

Having described of the ScheduleBuilder's sub-components, all that is missing is the ScheduleBuilder itself. A class diagram is included in figure 6.20, where the package private methods *schedule*, *unschedule*, and *reschedule* are invoked exclusively by the rewriters.

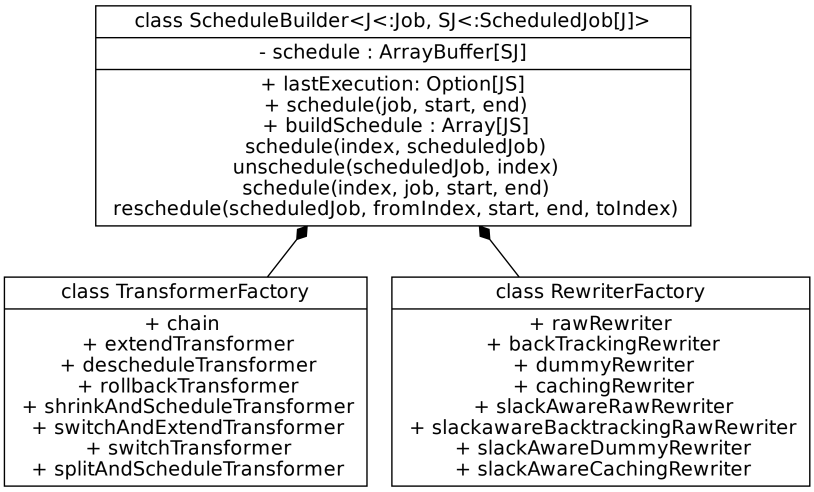

class ScheduleBuilder<J<:Job, SJ<:ScheduledJob[J]>
- schedule : ArrayBuffer[SJ]
+ lastExecution: Option[JS]
+ schedule(job, start, end)
+ buildSchedule : Array[JS]
schedule(index, scheduledJob)
unschedule(scheduledJob, index)
schedule(index, job, start, end)
reschedule(scheduledJob, fromIndex, start, end, toIndex)
class TransformerFactory
+ chain
+ extendTransformer
+ descheduleTransformer
+ rollbackTransformer
+ shrinkAndScheduleTransformer
+ switchAndExtendTransformer
+ switchTransformer
+ splitAndScheduleTransformer
class RewriterFactory
+ rawRewriter
+ backTrackingRewriter
+ dummyRewriter
+ cachingRewriter
+ slackAwareRawRewriter
+ slackawareBacktrackingRawRewriter
+ slackAwareDummyRewriter
+ slackAwareCachingRewriter

Figure 6.20: A class diagram describing the ScheduleBuilder and the factory classes.

A great deal of thought went into deciding what kind data structure would fit the Schedule schedule best. Although a linked list would provide O(1) read-write-modify requests, it would have to be a doubly linked list as schedule transversal can occur in any direction. In addition, it would also need to be a skip list to enable binary search within the list. Given this additional complexity, and because most operations are performed at the end of the schedule, an array was found to be a good compromise between simplicity, performance, and memory usage; as the doubly-linked skip list memory requirements can become a limiting factor when scheduling extremely long hyper periods.

Another interesting thing to note in figure 6.20 is the factory classes for the ScheduleBuilder's transformers and rewriters, this, and the fact that scala derives variable types from the invoked method, dismisses the need for long variable types with nested generic types.

6.2.6 ScheduleRewriter Package

The ScheduleRewriter is responsible for applying a transformation defined by a scheduler using one or multiple transformers. In addition, it provides an efficient interface to check if a transformation is valid before applying it. A ScheduleRewriter always implements the IScheduleBuilderRewriter trait, which contains a public method called *valid*, which returns true if the transformation is valid, and a package-private method called *execute*, which applies the transformation and can only be invoked by the ScheduleDecision.

Most transformations are a subclass of the abstract StandardRewriter class, which takes care of the apply/preview logistics while deferring the transformations sequence to its subclasses. The StandardRewriter contains a lazy boolean called valid, which simulates the transformation with a slack aware dummy rewriter, saving the slack tables for later use. If the apply method is invoked after a simulation has been performed, the transformation is applied using a raw rewriter and the previously computed slack tables; otherwise, a slack aware backtracking RawRewriter is used, which can recover from the scenario where the transformation is impossible. A class diagram including both entities is included in figure 6.21.

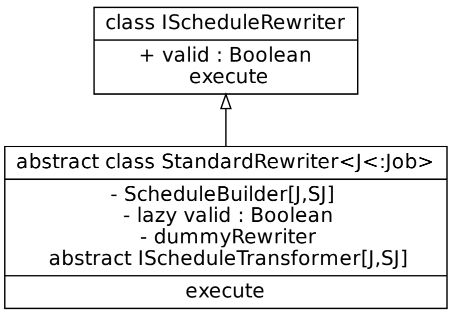


Figure 6.21: A class diagram describing the rewriter abstract classes.

Let's describe each ScheduleRewriter:

ExtendRewriter The extend rewriter extends the execution of a job, shifting other jobs to the right if necessary. The transformation is composed of a single extend transformer. The class is depicted in figure 6.22.

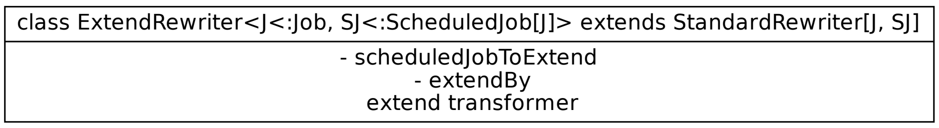
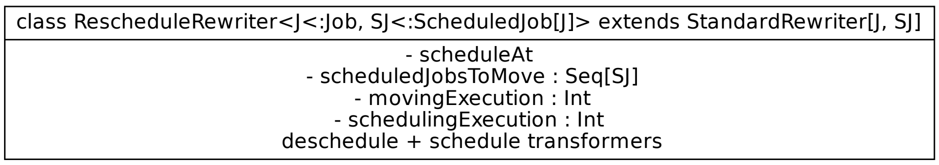

class ExtendRewriter<J<:Job, SJ<:ScheduledJob[J]> extends StandardRewriter[J, SJ]
- scheduledJobToExtend
- extendBy
extend transformer

Figure 6.22: A class diagram describing the ExtendRewriter class.

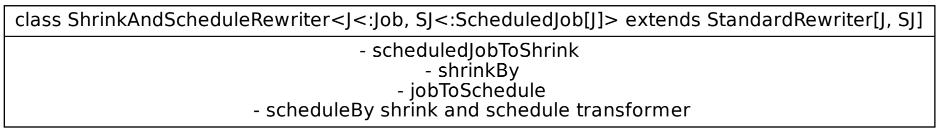
RescheduleRewriter The re-schedule rewriter combines multiple executions of a job at a given time. The rescheduler can schedule the execution past the length of all its constituent executions. The transformation is a sequence of a de-schedule transformer and a schedule transformer. A class diagram describing the class is included in figure 6.23.

ShrinkAndScheduleRewriter This rewriter shrinks an execution and pushes jobs backward, creating enough space to schedule a job non-preemptively. If an execution breaches its execution window, an attempt is made to exchange it with surrounding executions. The rewriter is composed of a single ShrinkAndScheduleTransformer. Figure 6.24 contains a class diagram for this class.



```
class RescheduleRewriter<J<:Job, SJ<:ScheduledJob[J]> extends StandardRewriter[J, SJ]
- scheduleAt
- scheduledJobsToMove : Seq[SJ]
- movingExecution : Int
- schedulingExecution : Int
deschedule + schedule transformers
```

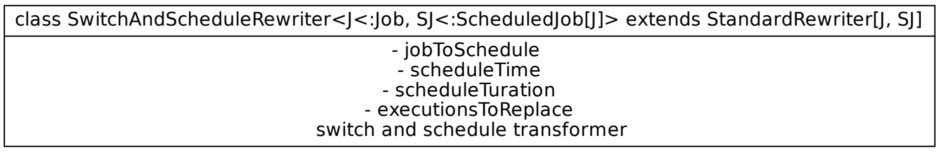
Figure 6.23: A class diagram describing the RescheduleRewriter class.



```
class ShrinkAndScheduleRewriter<J<:Job, SJ<:ScheduledJob[J]> extends StandardRewriter[J, SJ]
- scheduledJobToShrink
- shrinkBy
- jobToSchedule
- scheduleBy shrink and schedule transformer
```

Figure 6.24: A class diagram describing the ShrinkAndScheduleRewriter rewriter.

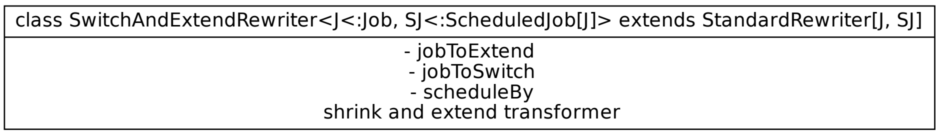
SplitAndScheduleRewriter The SplitAndScheduleRewriter splits an execution into two, scheduling one half in the past, thereby creating enough space to schedule one job non-preemptively. Jobs may be reordered as the transformer attempts to push jobs backward while keeping them within their execution windows. The transformation is composed of a single SplitAndScheduleTransformer. The class is illustrated in figure 6.25.



```
class SwitchAndScheduleRewriter<J<:Job, SJ<:ScheduledJob[J]> extends StandardRewriter[J, SJ]
- jobToSchedule
- scheduleTime
- scheduleTuration
- executionsToReplace
switch and schedule transformer
```

Figure 6.25: A class diagram describing the SplitAndScheduleRewriter class.

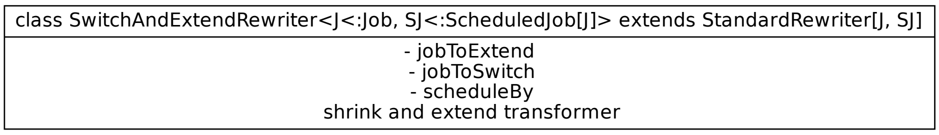
SwitchAndExtendRewriter Switches multiple executions of a job J_i by a job J_j, the executions are combined if any execution J_i neighbors J_j. The transformation is carried out by the SwitchAndExtendTransformer class. The class is depicted in figure 6.18.



```
class SwitchAndExtendRewriter<J<:Job, SJ<:ScheduledJob[J]> extends StandardRewriter[J, SJ]
- jobToExtend
- jobToSwitch
- scheduleBy
shrink and extend transformer
```

Figure 6.26: A class diagram describing the SwitchAndExtendRewriter class.

SwitchAndScheduleRewriter Switches the execution of two jobs using a switch transformer and schedules the switched job at a specified time. Figure 6.27 contains a class diagram for this class.



```
class SwitchAndExtendRewriter<J<:Job, SJ<:ScheduledJob[J]> extends StandardRewriter[J, SJ]
- jobToExtend
- jobToSwitch
- scheduleBy
shrink and extend transformer
```

Figure 6.27: A class diagram describing the SwitchAndScheduleRewriter class.

RollbackRewriter The RollbackRewriter is unique from all the other Rewriters presented here; it is the only one that does not extend StandardRewriter. The rollback rewriter rollsback the schedule until a specified time, and presents the possibility to schedule additional jobs through a *schedule* method.

Like the rewriters and transformers, a convenient factory named ScheduleRewriter is provided and accessible through the scheduleState. Figure 6.29 illustrates this entity through a class diagram.

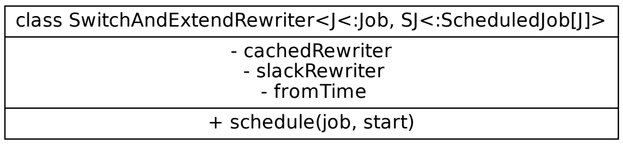


Figure 6.28: A class diagram describing the RollbackRewriter class.

To execute a re-write, the executeRewrite method must be invoked in the ScheduleDecision; a sequence diagram previewing the necessary instruction set is included in figure 6.30.

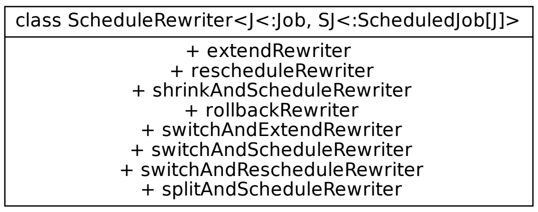

class ScheduleRewriter<J<:Job, SJ<:ScheduledJob[J]>
+ extendRewriter
+ rescheduleRewriter
+ shrinkAndScheduleRewriter
+ rollbackRewriter
+ switchAndExtendRewriter
+ switchAndScheduleRewriter
+ switchAndRescheduleRewriter
+ splitAndScheduleRewriter

Figure 6.29: A class diagram describing the SceheduleRewriter class.

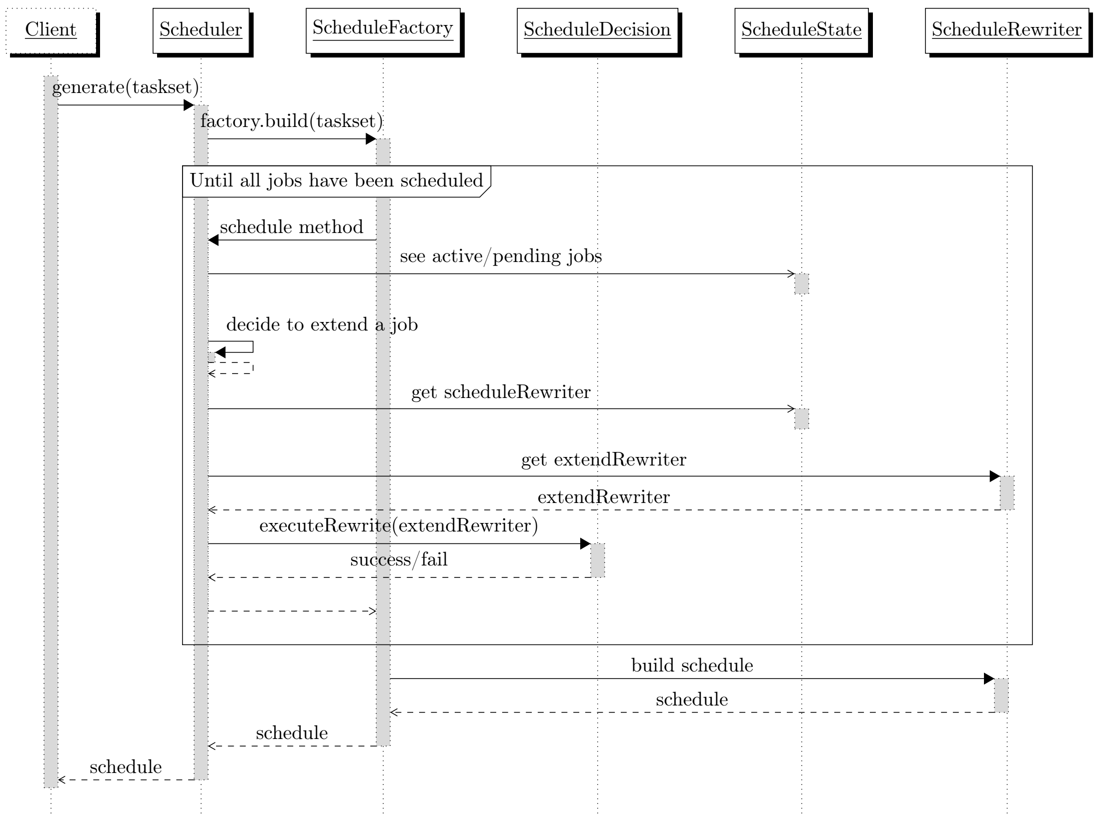


Figure 6.30: A sequence diagram for the main schedule loop.

6.2.7 Scheduler Package

The schedule package is home to all schedulers that use the framework. Every scheduler is defined by extending the IScheduler abstract class, which is composed of a protected ISchedule factory that is instantiated and configured by the extending scheduler to its desired configuration. The class also defines the generate methods invoked by the user to generate a schedule, which can accept a TaskSet object or an array of Tasks.

In addition to the IScheduler class, there are two extending traits to complement a schedule. An ILegacyScheduler provides a *legacyGenerate* method which generates a schedule without using the framework – useful when computing the slack of the schedule via a ReverseEDFScheduler. Finally, there is also the OrderedSchedule Trait, which defines a scheduler whose execution ordering follows a simple property which can be used to verify the correctness of its output. The OrderedScheduler trait defines the *verifyOrdering* function which returns true if the job schedule order follows the property defined by the scheduler. To give an example of such a property, the earliest deadline first scheduler only preempts when the deadline of the preempting job is larger or equal than the running job – the job has finished –, or when the preempting job is released later and has a shorter deadline – a higher priority release with a shorter deadline.

Figure 6.31 contains a class diagram for the Scheduler's package.

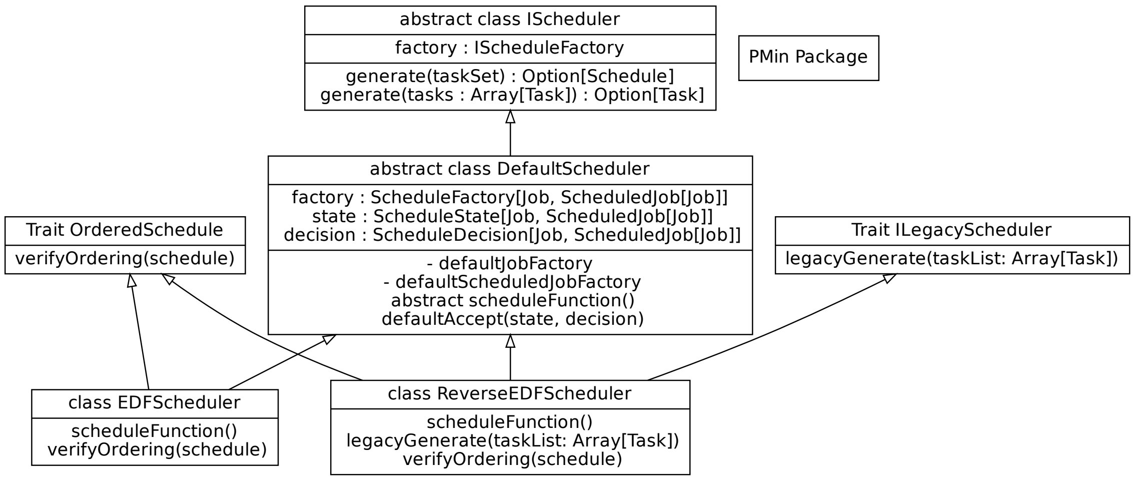

abstract class IScheduler
factory : IScheduleFactory
generate(taskSet) : Option[Schedule]
generate(tasks : Array[Task]) : Option[Task]
PMin Package
abstract class DefaultScheduler
factory : ScheduleFactory[Job, ScheduledJob[Job]]
state : ScheduleState[Job, ScheduledJob[Job]]
decision : ScheduleDecision[Job, ScheduledJob[Job]]
- defaultJobFactory
- defaultScheduledJobFactory
abstract scheduleFunction()
defaultAccept(state, decision)
Trait OrderedSchedule
verifyOrdering(schedule)
Trait ILegacyScheduler
legacyGenerate(taskList: Array[Task])
class EDFScheduler
scheduleFunction()
verifyOrdering(schedule)
class ReverseEDFScheduler
scheduleFunction()
legacyGenerate(taskList: Array[Task])
verifyOrdering(schedule)

Figure 6.31: A class diagram describing the Scheduler package.

Let's describe each entity in figure 6.31:

PMinPackage Contains an implementation of the scheduler developed in this thesis.

IScheduler An abstract class which all schedulers extend. Composed of a ScheduleFactory configured to the needs of the scheduler, and two generate methods.

DefaultScheduler Defines a standard scheduler which uses the standard job/scheduled job entities. The class provides the required job and scheduled job factories and the appropriate schedule factory configuration and instantiation such that any scheduler which requires this environment can simply extend DefaultScheduler and override the scheduleFunction() method.

EDFScheduler contains an implementation of the earliest deadline first schedule policy.

ReverseEDFScheduler defines an implementation of the reverse earliest deadline first scheduler.

OrderedSchedule Defines a schedule whose output contains a well-defined order which can be verified using the verifyOrdering function.

ILegacySchedule Characterizes a scheduler that provides a legacyGenerate method which produces a schedule without using the framework.

PMin Package

PMin – Preemption Minimizer – is the in-house name given to the scheduler presented in this thesis. Its class structure is divided into three main components: the PMinScheduler, the PMinActionController, and the PMinScheduleState.

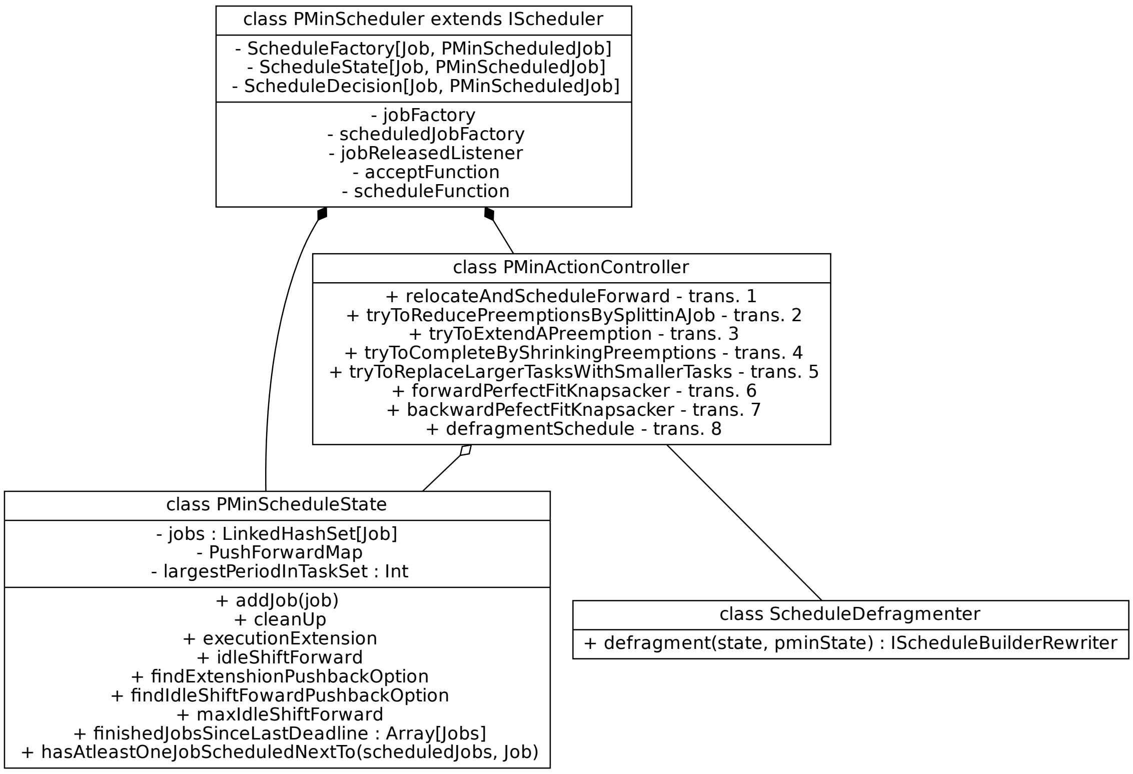
The PMinScheduler contains the main loop of the scheduling algorithm described in this document. In addition, the PMinScheduler class extends the IScheduler class and is responsible for configuring the schedule factory and all its required components. The class is structured like the pseudo-code presented in the final algorithm 5 of section 5.3.4, where several actions are called upon in a very specific order, but *how* those actions are implemented is delegated to the remaining components.

The PMinScheduleState tracks how many jobs have been extended, or moved forward – replaced by idle time. This is useful as we can query this database – instead of iterating the schedule – if we need to revert one of these transformations at a later time. In addition, it also contains methods to compute how much computation of a job can be shifted forward, or if a given job has any execution with another neighboring job J_i.

Finally, the PMinActionController is responsible for applying any transformation. Doing so may require knowing what is the *state* of the schedule, in which case it can query the PMinScheduleState component. In addition, it informs the PMinScheduleState as transformations are applied so that its database is always up to date.

The PMin scheduler is a well-structured component whose constituting elements have well-defined roles. In addition, the scheduler has a linear and simple call order, as PMinScheduler invokes the PMinActionController to attempt to apply the transformations described in, and in the order of, algorithm 5 in section 5.3.4. If the ActionController can apply a transformation which may need to be undone later, it will inform the PMinScheduleState which will add the transformation to its database until it expires – defined as the release time of the job the transformation was applied to, plus twice the longest period in the schedule. Failing to apply a transformation, the PMinScheduler will enact the two primary scheduling policies: (1) finish the largest job, or (2) execute the higher priority until the release of a higher priority release.

Figure 6.32 contains a class diagram for these components.



class PMinScheduler extends IScheduler
- ScheduleFactory[Job, PMinScheduledJob]
- ScheduleState[Job, PMinScheduledJob]
- ScheduleDecision[Job, PMinScheduledJob]
- jobFactory
- scheduledJobFactory
- jobReleasedListener
- acceptFunction
- scheduleFunction
class PMinActionController
+ relocateAndScheduleForward - trans. 1
+ tryToReducePreemptionsBySplittinAJob - trans. 2
+ tryToExtendAPreemption - trans. 3
+ tryToCompleteByShrinkingPreemptions - trans. 4
+ tryToReplaceLargerTasksWithSmallerTasks - trans. 5
+ forwardPerfectFitKnapsacker - trans. 6
+ backwardPefectFitKnapsacker - trans. 7
+ defragmentSchedule - trans. 8
class PMinScheduleState
- jobs : LinkedHashSet[Job]
- PushForwardMap
- largestPeriodInTaskSet : Int
+ addJob(job)
+ cleanUp
+ executionExtension
+ idleShiftForward
+ findExtenshionPushbackOption
+ findIdleShiftFowardPushbackOption
+ maxIdleShiftForward
+ finishedJobsSinceLastDeadline : Array[Jobs]
+ hasAtleastOneJobScheduledNextTo(scheduledJobs, Job)
class ScheduleDefragmenter
+ defragment(state, pminState) : IScheduleBuilderRewriter

Figure 6.32: A class diagram describing PMin's main components.

One entity present in the diagram which has not been described is the ScheduleDefragmenter, which contains an implementation of the schedule defragmentation algorithm presented in algorithm 4, particularly the defragment function as the forceSchedule function defined in the ScheduleUtils class of the ScheduleBuilder component. Another yet to describe entity is the PMinScheduledJob, which is the PMinScheduler extension of the ScheduledJob entity, and will be detailed further in this section.

Let's describe the class fields of each element in the diagram:

PMinScheduler Contains the schedule factory, which is configured with the required job factory and corresponding scheduledJobFactory – which instantiates PMinScheduledJobs, a subtype of ScheduledJob –, a job release listener which is invoked by the factory each time a job is released, an accept function which is invoked during the setup phase of the generate method to configure the correct schedule state and decision objects, and finally, the scheduleFunction which is invoked by the framework at each iteration.

PMinActionController Defines methods to attempt to apply each transformation in section 5.3.2.

PMinScheduleState the jobs LinkedHashSet[1] includes all active jobs that are currently released, or have been released two times the largest period in the past; the hash set is maintained by the cleanup function, which removes jobs that no longer respect this property. The PushForwardMap is another class which contains any moves that can effectively push another task forward, i.e., a preemption extension, or when jobs are moved forward, and replaced by idle time.

The executionExtension or the idleShiftForward methods are invoked whenever the PMinACtion controller performs an action which extends an execution, or moves multiple executions forward, replacing them by idle time. The find variants of these methods attempt to find these previous transformations to rollback if doing so reduces the total number of system preemptions.

The maxIdleShiftForward computes how much a job can be brought forward without breaking any deadlines; the finishedJobsSinceLastDeadline return an array of tasks which have been finished since the last deadline; and finally, the hasAtleastOneJobScheduledNextTo method checks if a list of executions as at least one which contains *job* has a neighbor.

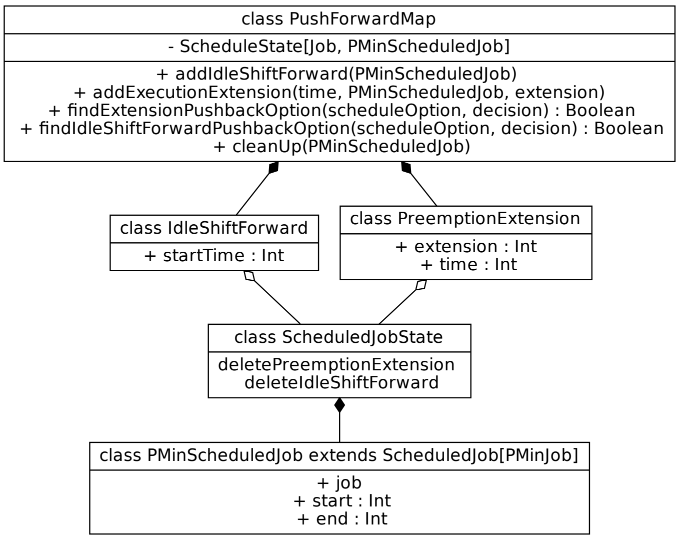

class PushForwardMap
- ScheduleState[Job, PMinScheduledJob]
+ addIdleShiftForward(PMinScheduledJob)
+ addExecutionExtension(time, PMinScheduledJob, extension)
+ findExtensionPushbackOption(scheduleOption, decision) : Boolean
+ findIdleShiftForwardPushbackOption(scheduleOption, decision) : Boolean
+ cleanUp(PMinScheduledJob)
class IdleShiftForward
+ startTime : Int
class PreemptionExtension
+ extension : Int
+ time : Int
class ScheduledJobState
deletePreemptionExtension
deleteIdleShiftForward
class PMinScheduledJob extends ScheduledJob[PMinJob]
+ job
+ start : Int
+ end : Int

Figure 6.33: A class diagram describing PMin's remaining components.

Figure 6.33 contains the remaining PMin entities. The PushForwardMap tracks any transformation that extends or completely moves an execution forward. It does so by having the addIdleShiftForward and addExecutionExtension methods invoked by the PMinActionController whenever it applies these transformations. Within the PushForwardMap, these transformations are tracked by the IdleShiftFoward and ExtensionShiftForward classes, each of them which is self-contained within a LinkedHashSet, which preserves insertion order but provides $O(1)$ removals on average. For convenience, the ScheduledJobState aggregates these objects within the ScheduledJobState, making them accessible from the PMinScheduledJob. The ScheduledJobState class was created to enforce a greater separation of the methods belonging to the ScheduledJob, and those of the scheduler.

When jobs are pushed towards preemption zones, PushForwardMap attempts to find an extension that can be *shrunk*, or, if no such execution exists, if another execution can be split in two, sending half of it backward. Ideally, an extension transformation is rolled back as it does not necessarily induce a preemption.

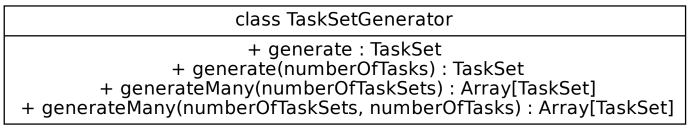
[1]A linked hash set preserves insertion order.

6.2.8 Util Package

The util package is composed of three entities: the Knapsacking algorithm, a random task set generator, a math util component to calculate the LCM of a given number set – required to calculate the hyper period of a task set –, and SortedUtils, which contains methods to operate sorted arrays via binary-search based methods. Let's describe each one of these packages in detail.

Random Task Set Generator

For the Random Task Set generator, we have used a python implementation of the randfixedsum algorithm [99] by Paul Emberson, Roger Stafford, and Robert Davis. The python script is included in the jar's resources and invoked as an external program through scala's ProcessBuilder. In the interest of access easiness, a TaskSetGenerator class provides seamless access to the generator. Figure 6.34 contains a class diagram describing the class.



```
class TaskSetGenerator
+ generate : TaskSet
+ generate(numberOfTasks) : TaskSet
+ generateMany(numberOfTaskSets) : Array[TaskSet]
+ generateMany(numberOfTaskSets, numberOfTasks) : Array[TaskSet]
```

Figure 6.34: A class diagram describing the TaskSetGenerator class.

Knapsacker

The Knapsacking problem is well defined and studied in the literature. It has been the topic of research for at least a century, as some works were published as early as 1897 [66]. The problem is defined as: given a set of items, each characterized by weight and value, determine which items to include in a collection such that the sum of the value of its constituent elements is maximized while its total weight is less or equal to a given limit.

A sub-problem of the Knapsack problem is called the subset-sum problem, where each item's value is equal to its weight. In our scheduler we have examples of both problems: (1) trying to fill a given space with as many executions as possible is a knapsack problem, as joining 3 executions of a job has a value of 3 but the weight – space needed to schedule those jobs – is equal to their combined execution time. Trying schedule as many jobs as possible non-preemptively within an execution window is a subset problem, since, the value of each execution is effectively equal to its weight.

The knapsack problem is a subset of the decision problem "Does a value exists that is at least V and does not exceed the maximum weight ? ". This problem is defined as NP-Complete, and as far as the authors of this thesis are aware, no known algorithm which is both optimal and fast exists to treat all cases [78]. The knapsack problem can be solved in a feasible amount of time only when the number of items is low, or when the absolute integer values of the item sets are low.

When the absolute integer value of weight is low, the problem can be solved in pseudo-polynomial time through dynamic programming [84]. Dynamic algorithms solve complex problems by breaking them down into a sub-collection of simpler problems and re-using their solutions. In this case, the algorithm would solve all Knapsack problems starting with a weight of 0 up to a weight of N; at each iteration, the algorithm would reuse previously built solutions to compute a valid solution for $W + 1$.

When the number of items is low, the knapsack problem can be solved efficiently using meet-in-the-middle [44]. This approach splits items into two sorted sets and generates all possible combinations within these sets in $O(2^{n/2})$. Finally, for each combination in the first set, find the largest element in the second set such that the sum of both is smaller or equal than the weight and its value is maximized. Using binary search to find the best match, the worst-case run-time becomes $O(n \times 2^{n/2})$, with a space requirement of $O(2^{n/2})$.

More complex solutions involve a hybridization of these approaches [9]. Given their high complexity, they are beyond the scope of this thesis. Because the computational requirements of each execution can be very big, meet-in-the-middle is likely to be more efficient. In this chapter, we describe a parallel version of the meet-in-the-middle algorithm.

The solution is based on parallelizing the combinatorial generation process and the subsequent search of this space. The combinatorial generation process is split into multiple jobs where each is responsible for generating all combinations of N items in one of the two sets. Then, the research space on one set is divided into pieces using a java *spliterator*. The sorted search space which can only contain sub-optimal solutions than the current best or invalid solutions is continuously pruned each time the set is split into two by the spliterator. Finally, for each divided set, the algorithm searches the research space in parallel until a matching element is found using the *floor* operation of the set.

While the search phase is read-only and can be easily be performed in parallel, the first phase where all combinations of $n/2$ elements are generated is not. To perform the combinatorial generation in parallel, a ConcurrentSkipList is used. A skiplist, first describe in [83], is an ordered data structure that permits $O(log\ n)$ average performance on search, delete, and insert operations. It is a probabilistic linked-list based data structure composed of multiple levels, where the last level contains all elements, and each level acts as "*express lanes*" for the lane below. During insertion, a "*coin toss*" decides whether an element is present on a level, hence being a probabilistic data-structure. The Java ConcurrentSkipListSet was developed by Lea [59].

One particularity of our Knapsack usage is that not only must all items respect the maximum weight, they must also be *schedulable*. If a particular set is not schedulable because its execution bounds are mutually exclusive, then that combination is useless and should immediately be eliminated. Hence, the operation that combines multiple items into one must be user-defined. Because of this, not only is the combination a user definable object but so is the item type to knapsack as well; hence like the scheduler, these types are user-defined.

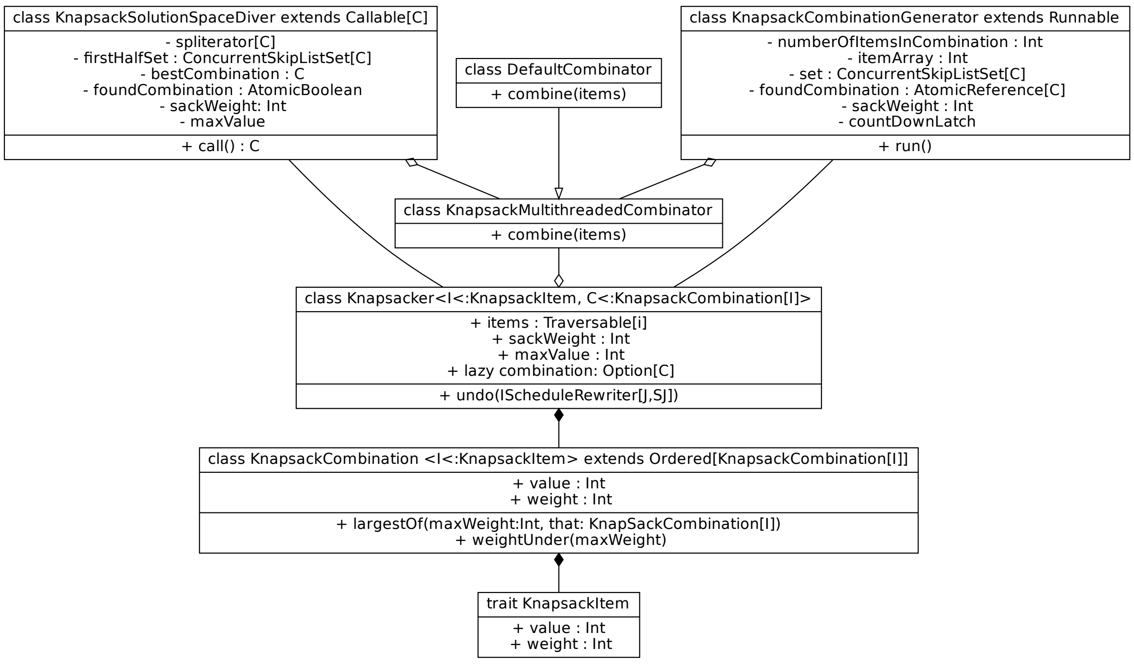

class KnapsackSolutionSpaceDiver extends Callable[C]
- spliterator[C]
- firstHalfSet : ConcurrentSkipListSet[C]
- bestCombination : C
- foundCombination : AtomicBoolean
- sackWeight: Int
- maxValue
+ call() : C
class DefaultCombinator
+ combine(items)
class KnapsackCombinationGenerator extends Runnable
- numberOfItemsInCombination : Int
- itemArray : Int
- set : ConcurrentSkipListSet[C]
- foundCombination : AtomicReference[C]
- sackWeight : Int
- countDownLatch
+ run()
class KnapsackMultithreadedCombinator
+ combine(items)
class Knapsacker<I<:KnapsackItem, C<:KnapsackCombination[I]>
+ items : Traversable[i]
+ sackWeight : Int
+ maxValue : Int
+ lazy combination: Option[C]
+ undo(IScheduleRewriter[J,SJ])
class KnapsackCombination <I<:KnapsackItem> extends Ordered[KnapsackCombination[I]]
+ value : Int
+ weight : Int
+ largestOf(maxWeight:Int, that: KnapSackCombination[I])
+ weightUnder(maxWeight)
trait KnapsackItem
+ value : Int
+ weight : Int

Figure 6.35: A class diagram describing the Knapsacker Package.

A class diagram for the Knapsacker package is provided in figure 6.35, where :

KnapsackItem A user extendable trait that defines an item to be knapsacked. Only two methods exist, one to compute its value, and another its weight.

KnapsackCombination Represents a combination of KnapsackItems as defined by the user. The class is extendable, permitting the user to accommodate custom types, which can be built by a user-defined KnapsackMultithreadedCombinator, if required.

Knapsacker The main Knapsacker class to invoke the knapsacker. The user instantiates the Knapsacker with the items whose knapsacking is desired, the max value and weight of the sack. The knapsacker is invoked by calling the lazy combination method, which returns a combination if one exists.

KnapsackMultithreadedCombinator Defines a KnapsackCombinator that can combine items of type I into KnapsackCombinations of type C. The object must support parallel invocation of the combine method.

DefaultCombinator Defines a default combinator which merely combines all items into an array.

KnapsackCombinatorGenerator Responsible for generating all combinations of length N of a given item array. When all combinations have been computed, or an optimal solution has been found, a countdown latch is updated to notify the Knapsacker the operation is complete.

KnapsackSolutionSpaceDiver Tries to find the optimal combination on a slice of the set, returning the best result within it or quitting if the best result has been found by any space diver.

A sequence diagram demonstrating the interactions between these classes is provided in figure 6.36.

Math Utils

The math utils class, like the name suggests, contains math related functions. The class is composed of two methods, one that computes the least common denominator (LCM) of two integers, and another which uses this method to compute the LCM of N integers. Figure 6.37 contains a class diagram for this class.

Sorted Utils

Sorted Utils contains methods to efficiently search, insert, and remove items from sorted arrays using binary search based algorithms. The class requires an *Ordering* type to be able to compare any class with a natural ordering. A class diagram depicting SortedUtils is included in figure 6.38.

binarySearchClosest searches for the closest element matching K using binary search.

binarySearch searches for the element K using binary search.

binarySearchInsertion Inserts an element into an array using binary search to resolve its index.

binarySearchRemoval Removes an element from a SortedArray using binary search to resolve its index.

6.3 Validation and Testing

Given the research-oriented nature of the project, it is imperative that the methods proposed here be correct. This is especially important when considering the proposed methodology to compute slack, as no formal proof of their correctness is provided.

As stated in the planning section, testing is performed through the JUnit testing framework, ensuring java interoperability. However, because some classes can only be used by native scala code, scalatest is required to test these components. This is less than ideal, as it demands the tester to work with two different testing frameworks. Luckily, scalatest is compatible with the JUnit runner, and scalatest classes can be invoked by the JUnit running by appending the *@runWith(classOf[JUnitRunner])* class tag to any scalatest test class. Hence, although two different testing frameworks are actively used, only the JUnit runner is required to launch all tests.

The test methodology is separated into three categories:

1. Unit testing, which tests a specific component in isolation.

2. Component Testing, which tests a specific component that can be comprised of many subcomponents, or/and uses the framework to generate the necessary conditions for that test, i.e., job instantiation, slack generation, or a predefined scheduling order.

3. Automated Property Testing, which tests a given class or component by automatically generating test cases and verifying the output through a given property.

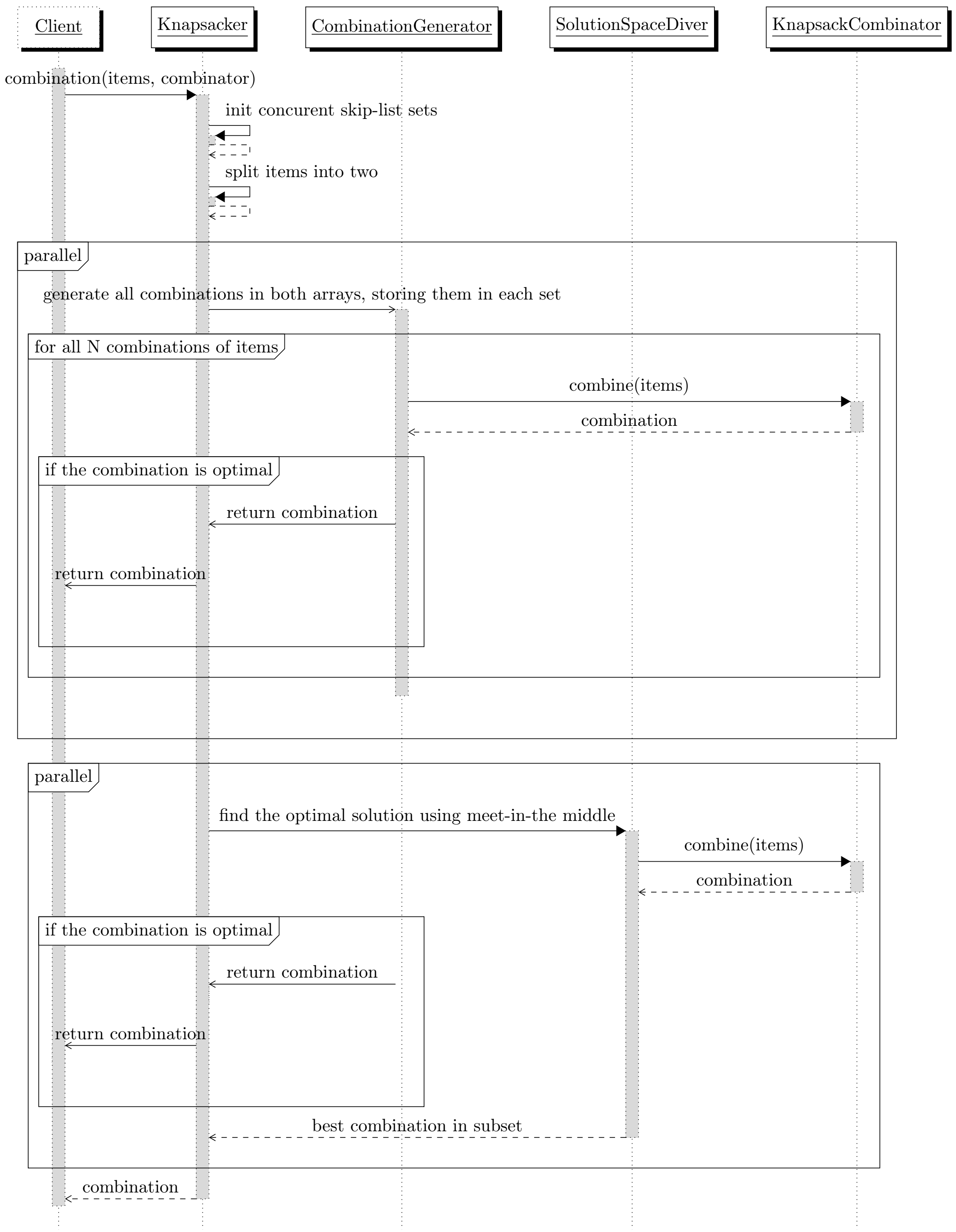

Client
Knapsacker
CombinationGenerator
SolutionSpaceDiver
KnapsackCombinator
combination(items, combinator)
init concurent skip-list sets
split items into two
parallel
generate all combinations in both arrays, storing them in each set
for all N combinations of items
combine(items)
combination
if the combination is optimal
return combination
return combination
parallel
find the optimal solution using meet-in-the middle
combine(items)
combination
if the combination is optimal
return combination
return combination
best combination in subset
combination

Figure 6.36: A sequence diagram for the main schedule loop.

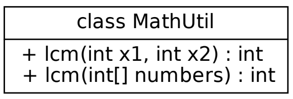
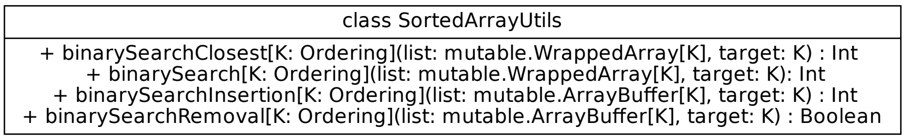


Figure 6.37: A class diagram describing the MathUtil class.

To facilitate automatic property testing, an additional testing framework, *scalacheck*, was used. Scalacheck provides the necessary architecture to define automatic testing, namely random data generators and the class structure to do so. Each automatic test was executed for at least 24 hours, providing a high degree of confidence the framework and its schedulers are bug-free.

In addition to all included tests, an extensive use of assertions and requirements also provides a



class SortedArrayUtils

+ binarySearchClosest[K: Ordering](list: mutable.WrappedArray[K], target: K) : Int
+ binarySearch[K: Ordering](list: mutable.WrappedArray[K], target: K): Int
+ binarySearchInsertion[K: Ordering](list: mutable.ArrayBuffer[K], target: K) : Int
+ binarySearchRemoval[K: Ordering](list: mutable.ArrayBuffer[K], target: K) : Boolean

Figure 6.38: A class diagram describing the SortedUtils class.

high error resiliency, as a scheduling error leading to an illegal action is sure to be detected. These assertions, too fine-grained to enumerate, include: all jobs obey their release and deadline bounds, all jobs are scheduled in accordance with their computational requirements, only one job executes at any given time, and no slacks are negative.

The following subsections will describe the testing and validation of all components, sub-components, and proposed algorithms.

Slack Calculators

The Slack Calculator is the most extensively tested component in the entire project. This is well justified, as a slack computation error can lead to unexpected and hard to understand circumstances. Most of the times, the error will go unnoticed until only a few cycles later, where it will likely incur a deadline miss, causing the scheduler to crash. Alternatively, the error can also go completely unnoticed when the error underflows the available slack, or the scheduler does not attempt to maximize the slack of the misbehaving job.

To ensure the correct functioning of the slack mechanism, the combinations of all possible interactions between two jobs have been enumerated and tested; this accounts to 15 scenarios in which the interweave between two jobs is slid from the left to the right, in addition to all combinations of reverse slack when applicable. Additionally, we have also considered how the mathematical method is implemented and developed test cases with:

1. Multiple carry out jobs, including nested scenarios.
2. Multiple idle-period shadowing tests, including nested scenarios.
3. Jobs which are considered in both equations 5.12 and 5.11, described in section 5.2.2 on page 33.

The number of unit tests on the slack computation module alone, not including sub-formulas, totals 31 tests. For each test, the slack is manually computed for the whole hyper-period, or until the first deadline of the longest task in the schedule if the hyper-period is too long. The produced slack values are cross-checked with all 3 implementations of the slack calculator. Finally, the task set is scheduled with the reverse earliest deadline scheduler, which will postpone each job for as long as possible, and will crash if one of the values is higher than it should be.

The sub-formulas defined in section 5.2.2, namely: calculating the overlap between two jobs – equation 5.14, if one of job J_i carries out another job J_j, if a job J_i contained within another job J_j, the first release of a task after a specified time – equation 5.6, and the last deadline of a task before or at a given time – equation 5.8, have also been unit tested. These tests take all required values within the tasks, i.e., job deadline, task release, task period, etc., and produce all possible combination between them, testing all possible combinations.

Finally, the three scheduler implementations have been tested using automated property testing. These tests are composed of randomly generated task sets composed of 1 up to 500 jobs, where all three generators must agree with the slack value of all jobs. In addition, the reverse earliest deadline scheduler will schedule the taskset, guaranteeing that if one of these values is higher than the correct value, the scheduler will crash.

ScheduleBuilder

The ScheduleBuilder is tested in all but the slack tests, as it the component responsible for managing its output – the schedule. Because of this, the only *unit tests* in this component are reserved to the Rewriter classes, as the rewriting transformations are tested as part of the *ScheduleRewriter* component. In this regard, the RawRewriter has each defined operation tested. The backtracking rewriter is also extensively tested, where for each operation, multiple operations are performed to try and induce an error.

ScheduleJobTree

The ScheduleJobTree has also been extensively tested. The component tests verify all possible combinations between two jobs, sliding from the left to the right; for each combination, the execution order is permuted, and the slack/correct release of all jobs is verified. The tests are performed via a custom-built scheduler with pre-defined actions that verifies the complete ScheduleState object at every iteration.

Finally, the ScheduleJobTreeRewriter tests are also extensive; as each operation is tested in two conditions: one in which the slack of the job is affected, and one in which it suffers no change.

Schedule Rewriter

The rewriter tests are comprised of a single test for each available rewrite operation. Although a single test for each transformation may not encompass a significant portion of all possible scenarios, these tests are complemented by PMin scheduler tests, where multiple transformations are applied in sequence.

Schedulers

Both the EDFScheduler and the ReverseEDFScheduler have been unit tested through 10 manually verified task sets. In addition to these unit tests, automated property tests are also employed and verified through the respective OrderedSchedule's properties. All these tests also verify that the produced schedule is valid.

The PMinScheduler is verified through 25 task sets which have been manually verified by hand to ensure all transformations are correctly applied. Complementing these tests, are 14 unit tests that have been specially crafted to test all transformations both individually and in sequence. Finally, automated property testing is employed to verify that each produced schedule is valid, i.e., all executions are allocated within their release and deadline bounds, and all jobs have their computational requirements satisfied.

Knapsacker

The Knapsacker is composed of six manual tests that have been carefully designed considering the algorithm and its respective implementation. The tests are designed such that all the following cases are covered:

1. The value and weight are such that no element can be combined.
2. The optimal value is generated during the generation of one set.
3. The optimal value is the combination of all elements.
4. The value is a combination of two elements in each set.
5. A custom combinator is used.
6. A combination of items which is below the optimal value.

In addition to these unit tests, automated property tests are used to provide a high degree of confidence the knapsacker works correctly. The tests generate an array initialized with random data, pick a random combination of N numbers, and ensure the knapsacker is able to find this combination.

Chapter 7

Conclusion

In this thesis, we have approached the problem of minimizing the number of preemptions within ARINC-653 certified avionic systems. In ARINC, context switches can extend the worst-case execution time of a partition up to 33% [18]; this increases the overall number of computing nodes required to perform a computation, ultimately leading to increased gross weight, fuel costs, and CO_2 emissions [106]. Although many schedulers in the state-of-the-art are able to produce a feasible schedule for avionic systems, none attempt to minimize the number of system preemptions.

With the goal of minimizing preemptions and maximizing efficiency, we have produced two contributions. The first is a new exact and precise slack mechanism that is efficient and completely independent of the active schedule policy, providing a scheduler with complete freedom of choice when deciding which task to schedule next. The slack mechanism is able to significantly prune the research space by cutting invalid branches – where at least one task misses its deadline – at their roots, thereby preventing expensive look ups that can only lead to invalid solutions. The mechanism is composed of two phases, the first is a slack extraction phase that can be performed through a mathematical or algorithmic approach; and a second phase, called the scheduling phase where at each iteration, the scheduler decides which task to schedule next and the slack of all the impacted tasks is updated.

The second contribution is a scheduler capable of minimizing the number of system preemptions. The scheduler is based on a set of greedy heuristics that generally provide a good solution, but incorporates a fall-back recovery mechanism that is triggered whenever the heuristics prove to be sub-optimal. Given that no scheduler currently attempts to minimize the number of system preemptions, we have chosen to compare our solution against two state-of-the-art non-preemptive offline schedulers. Given that these schedulers are specifically tailored for the non-preemptive domain, it is unrealistic to expect our solution can beat them. Although our method does not improve significantly upon the state-of-the-art in terms of non-preemptive schedulability ratio, it offers comparable performance to some of these methods. In task sets that are failed to be scheduled non-preemptively, most jobs (98% to 99%) are scheduled non-preemptively, proving our method is indeed very good in reducing the number of system preemptions.

The project's success not only contributed to the advancement of scheduling theory and real-time safety-critical systems such as those used in aviation, but also CISTER's position as a leading research unit in the area of real-time & embedded platforms, and was without doubt exceedingly ambitious and challenging. Of course, none of the work would be possible if it was not for my supervisors Geoffrey Nellissen and Eduardo Tovar, which while providing me with complete directional freedom, were instrumental in nurturing and shaping this work to excellence.

Appendix

A.1 Algorithms

Reverse EDF Algorithm

Algorithm 6 Reverse EDF Scheduler.

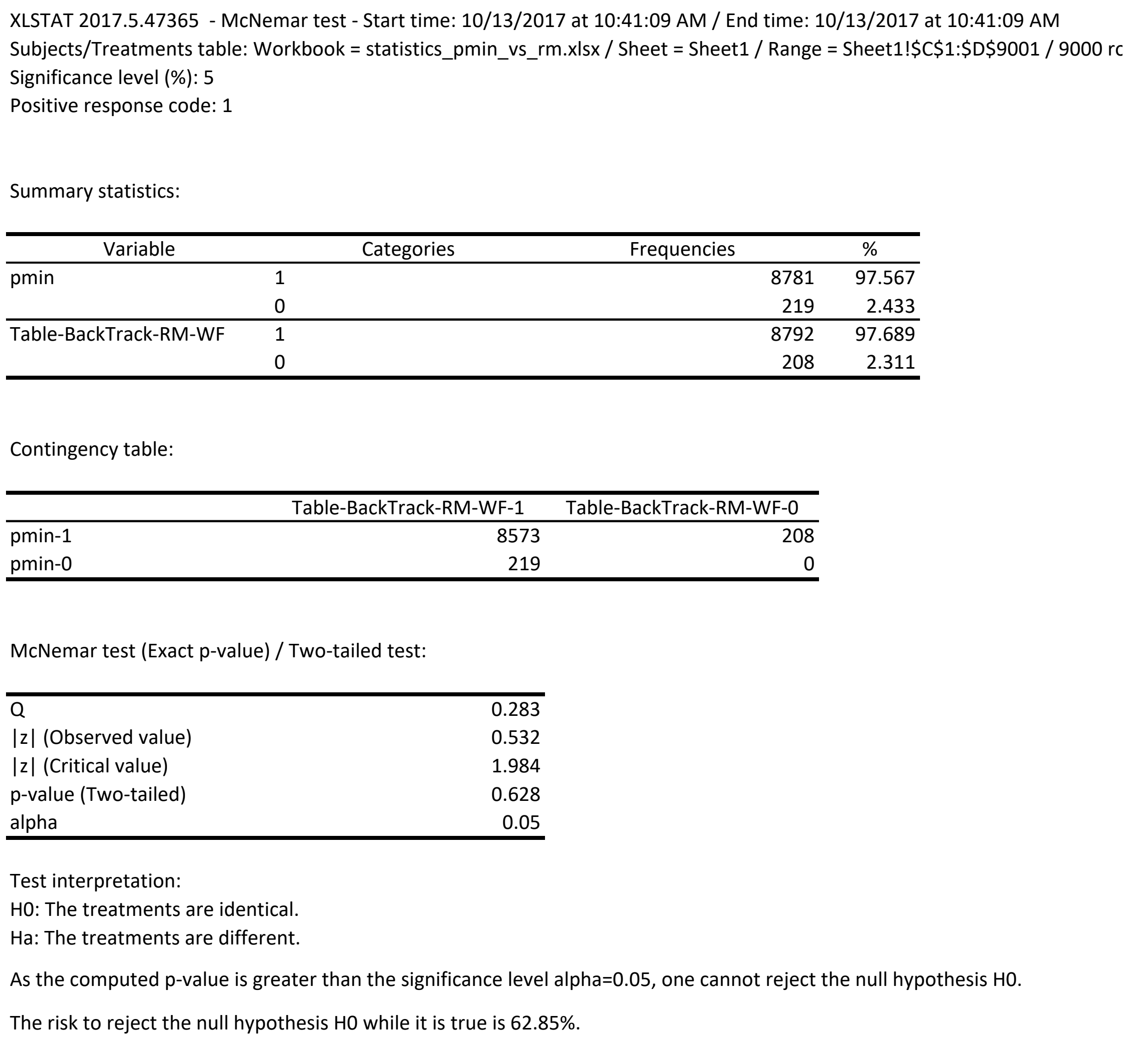
```
function SCHEDULE($task\ set$)
    $job\ list \leftarrow task\ set$.jobs
    sort $job\ list$ by ($job \Rightarrow -job$.deadline, $-job$.release, $job.task$.id)    ▷ Sort in Reverse EDF
    $schedule \leftarrow$ empty list
    $time \leftarrow job\ list$.first.deadline
    repeat
        $time \leftarrow$ min($head\ job$.deadline, $time$) ▷ Determine the earliest time there are active jobs
        $job \leftarrow job\ list$.iterator
                .takeWhile(_.deadline $\geq time$)
                .maxBy($job \Rightarrow job$.release, $-\ job$.deadline)    ▷ Get the active job with the latest
  release, using earliest deadline as a tie breaker

        $runtime \leftarrow job\ list$.iterator
                .dropWhile(_.deadline $\geq time$)
                .takeWhile(_.deadline $> time -$ job.execution)
                .find($iJob \Rightarrow iJob$.release $>$ $job$.release or $iJob$.release $=$ $job$.release and
  $iJob$.deadline $<$ $job$.deadline)
                .getOrElse(min($time -$ _.deadline, $job$.execution), $job$.execution)    ▷ Calculate
  runtime when a higher priority job is released within the execution of Job

        $time \leftarrow time - runtime$
        $job$.runtime $\leftarrow job$.runtime $- runtime$
        add execution of $job$ for $runtime$ units to the front of $schedule$
        if $job$.finished then
            remove $job$ from $job\ list$
        end if
    until $job\ list$.empty
end function
```

Algorithm 6 schedules jobs from the end of the hyper period, under latest release and earliest deadline priority order. It has a time complexity of $O(J \times \log_J + E \times (T + T))$, where J is the number of system jobs and E is the number of total executions. $J \log_j$ is the cost of sorting the job list in reverse EDF order, and $E \times (T + T)$ the price of, for each execution, searching for the job to schedule plus resolve if a higher priority job is released during its execution.

A.2 Statistical Tests

Schedulability Analysis

PMin vs CWin-RM



XLSTAT 2017.5.47365 - McNemar test - Start time: 10/13/2017 at 10:41:09 AM / End time: 10/13/2017 at 10:41:09 AM
Subjects/Treatments table: Workbook = statistics_pmin_vs_rm.xlsx / Sheet = Sheet1 / Range = Sheet1!$C$1:$D$9001 / 9000 rc
Significance level (%): 5
Positive response code: 1

Summary statistics:

Variable	Categories	Frequencies	%
pmin	1	8781	97.567
	0	219	2.433
Table-BackTrack-RM-WF	1	8792	97.689
	0	208	2.311

Contingency table:

	Table-BackTrack-RM-WF-1	Table-BackTrack-RM-WF-0
pmin-1	8573	208
pmin-0	219	0

McNemar test (Exact p-value) / Two-tailed test:

Q	0.283
|z| (Observed value)	0.532
|z| (Critical value)	1.984
p-value (Two-tailed)	0.628
alpha	0.05

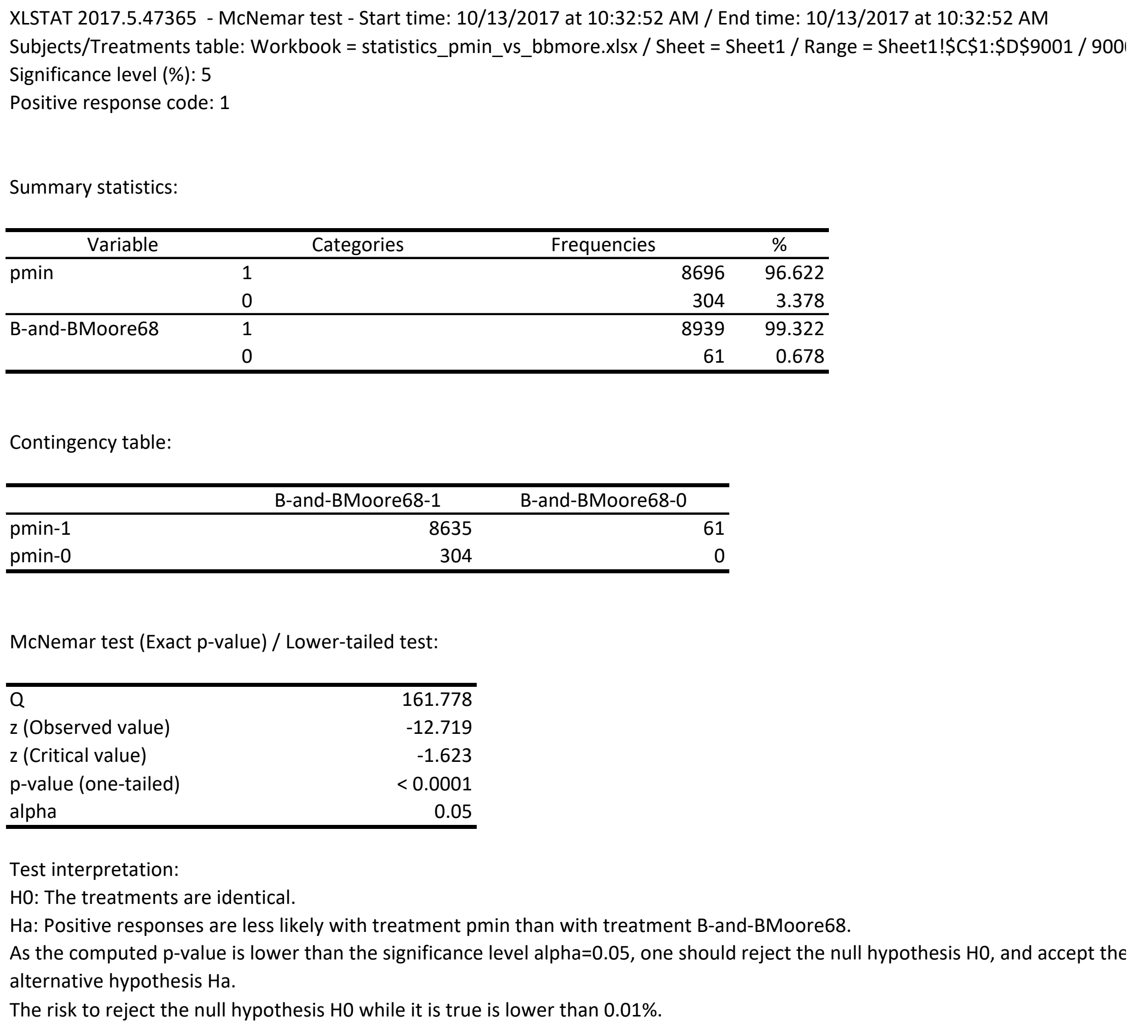
Test interpretation:
H0: The treatments are identical.
Ha: The treatments are different.

As the computed p-value is greater than the significance level alpha=0.05, one cannot reject the null hypothesis H0.

The risk to reject the null hypothesis H0 while it is true is 62.85%.

Figure A.1: A McNemar test which fails to reject the hypothesis PMin and CWin-RW present different schedulability ratios on 9000 non-preemptive task sets.

PMin vs BB-Moore



XLSTAT 2017.5.47365 - McNemar test - Start time: 10/13/2017 at 10:32:52 AM / End time: 10/13/2017 at 10:32:52 AM
Subjects/Treatments table: Workbook = statistics_pmin_vs_bbmore.xlsx / Sheet = Sheet1 / Range = Sheet1!$C$1:$D$9001 / 900
Significance level (%): 5
Positive response code: 1

Summary statistics:

Variable	Categories	Frequencies	%
pmin	1	8696	96.622
	0	304	3.378
B-and-BMoore68	1	8939	99.322
	0	61	0.678

Contingency table:

	B-and-BMoore68-1	B-and-BMoore68-0
pmin-1	8635	61
pmin-0	304	0

McNemar test (Exact p-value) / Lower-tailed test:

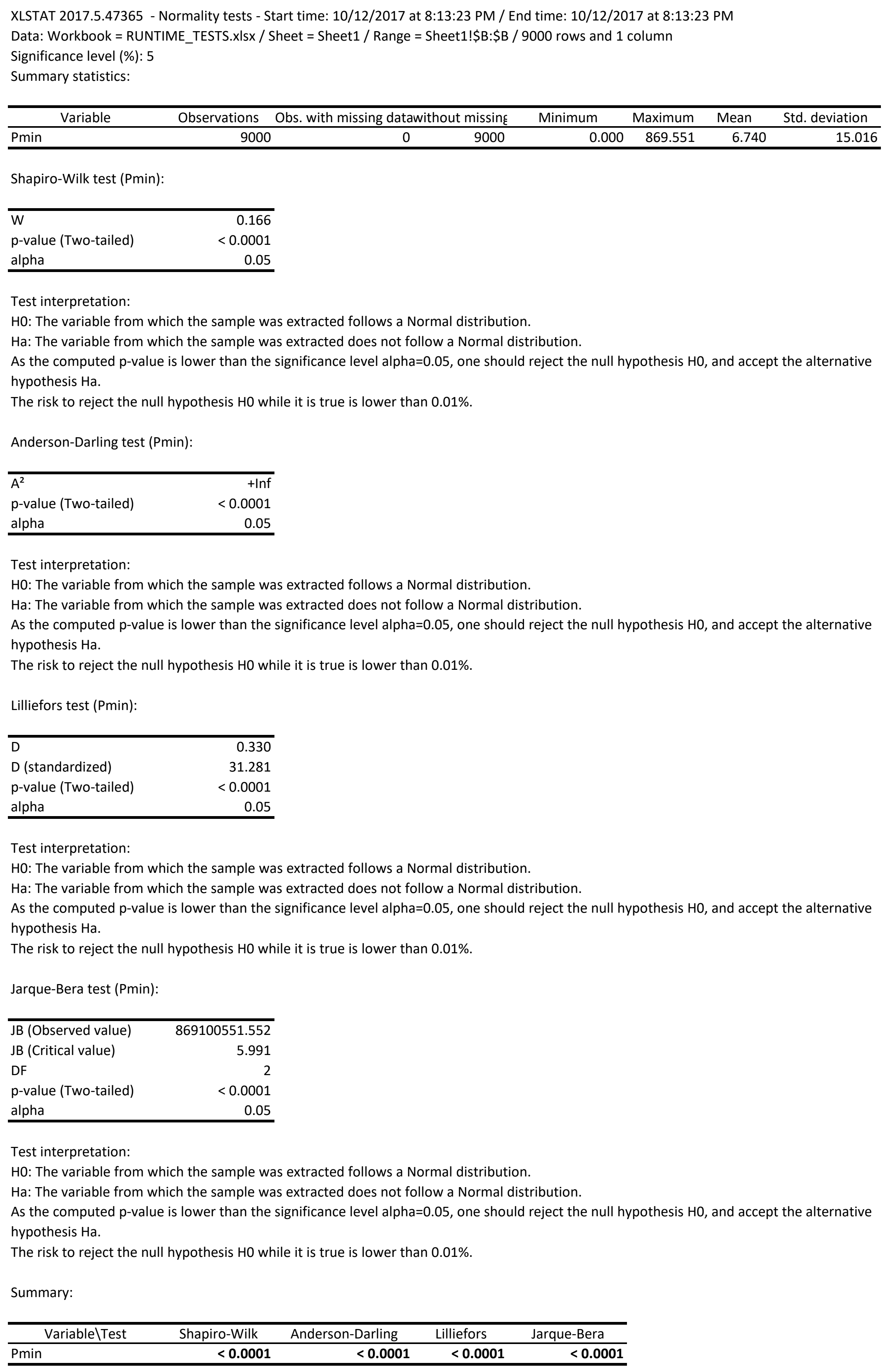
Q	161.778
z (Observed value)	-12.719
z (Critical value)	-1.623
p-value (one-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The treatments are identical.
Ha: Positive responses are less likely with treatment pmin than with treatment B-and-BMoore68.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Figure A.2: A McNemar test which suggests there is enough evidence to assume BB-Moore has a higher schedulability ratio than PMin.

Runtime Analysis

PMin Normality



XLSTAT 2017.5.47365 - Normality tests - Start time: 10/12/2017 at 8:13:23 PM / End time: 10/12/2017 at 8:13:23 PM
Data: Workbook = RUNTIME_TESTS.xlsx / Sheet = Sheet1 / Range = Sheet1!$B:$B / 9000 rows and 1 column
Significance level (%): 5
Summary statistics:

Variable	Observations	Obs. with missing data	without missing	Minimum	Maximum	Mean	Std. deviation
Pmin	9000	0	9000	0.000	869.551	6.740	15.016

Shapiro-Wilk test (Pmin):

W	0.166
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Anderson-Darling test (Pmin):

A²	+Inf
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Lilliefors test (Pmin):

D	0.330
D (standardized)	31.281
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Jarque-Bera test (Pmin):

JB (Observed value)	869100551.552
JB (Critical value)	5.991
DF	2
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Summary:

Variable\Test	Shapiro-Wilk	Anderson-Darling	Lilliefors	Jarque-Bera
Pmin	**< 0.0001**	**< 0.0001**	**< 0.0001**	**< 0.0001**

Figure A.3: Multiple statistic tests which suggest PMin's execution times do not follow a normal distribution.

CWin-RM Normality

XLSTAT 2017.5.47365 - Normality tests - Start time: 10/12/2017 at 8:25:54 PM / End time: 10/12/2017 at 8:25:55 PM
Data: Workbook = RUNTIME_TESTS.xlsx / Sheet = Sheet1 / Range = Sheet1!$A:$A / 9000 rows and 1 column
Significance level (%): 5
Summary statistics:

Variable	Observations	Obs. with missing dat	without missin	Minimum	Maximum	Mean	Std. deviation
CW-RM	9000	0	9000	2.000	58859.000	105.469	1040.423

Shapiro-Wilk test (CW-RM):

W	0.025
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Anderson-Darling test (CW-RM):

A²	+Inf
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Lilliefors test (CW-RM):

D	0.461
D (standardized)	42.023
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Jarque-Bera test (CW-RM):

JB (Observed value)	1518074458.839
JB (Critical value)	5.991
DF	2
p-value (Two-tailed)	< 0.0001
alpha	0.05

Test interpretation:
H0: The variable from which the sample was extracted follows a Normal distribution.
Ha: The variable from which the sample was extracted does not follow a Normal distribution.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.

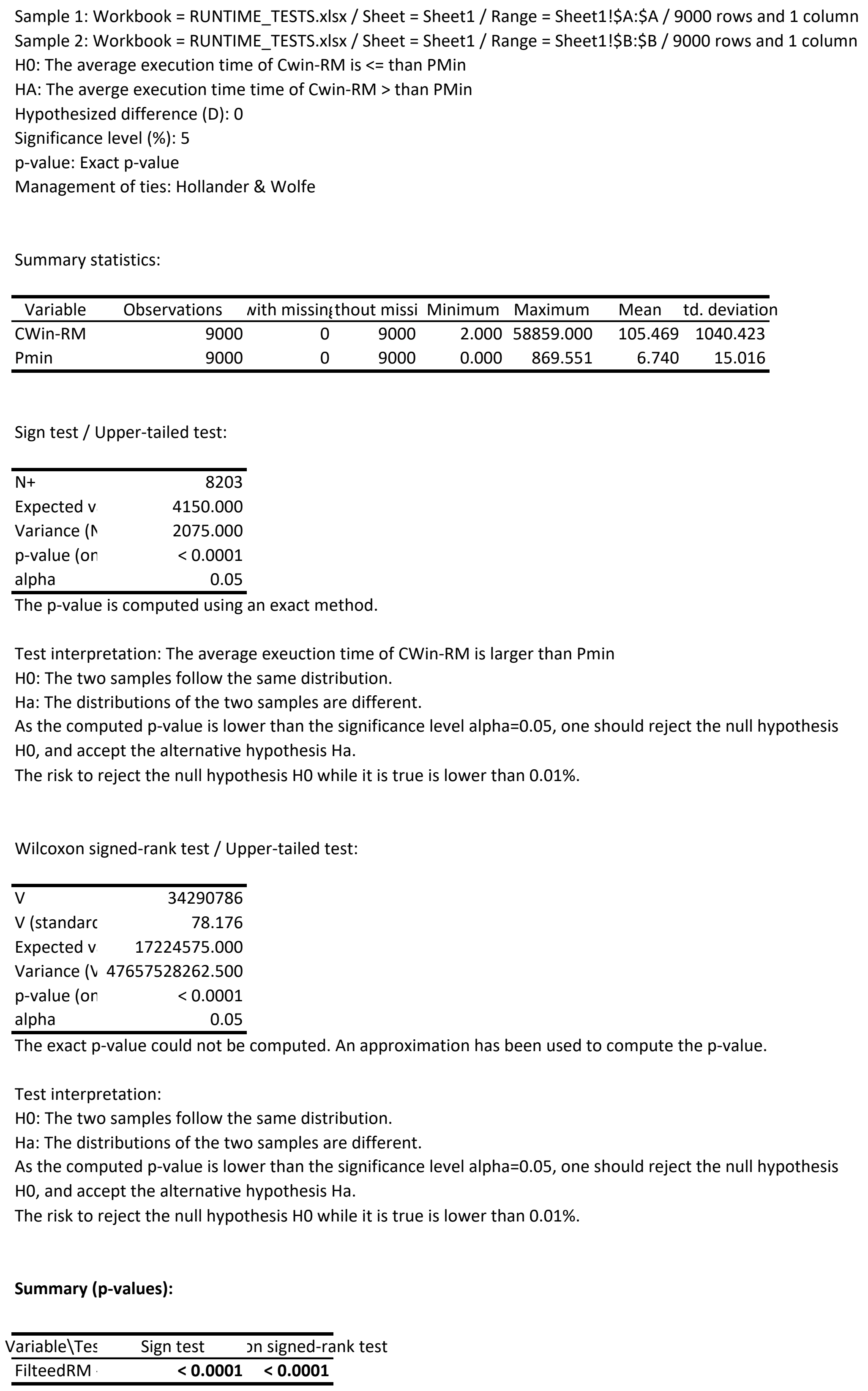
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Summary:

Variable\Test	Shapiro-Wilk	Anderson-Darling	Lilliefors	Jarque-Bera
CW-RM	**< 0.0001**	**< 0.0001**	**< 0.0001**	**< 0.0001**

Figure A.4: Multiple statistic tests which suggest CWin-RM's execution times do not follow a normal distribution.

PMin vs CWin-RM



Sample 1: Workbook = RUNTIME_TESTS.xlsx / Sheet = Sheet1 / Range = Sheet1!$A:$A / 9000 rows and 1 column
Sample 2: Workbook = RUNTIME_TESTS.xlsx / Sheet = Sheet1 / Range = Sheet1!$B:$B / 9000 rows and 1 column
H0: The average execution time of Cwin-RM is <= than PMin
HA: The averge execution time time of Cwin-RM > than PMin
Hypothesized difference (D): 0
Significance level (%): 5
p-value: Exact p-value
Management of ties: Hollander & Wolfe

Summary statistics:

Variable	Observations	vith missin	thout missi	Minimum	Maximum	Mean	td. deviation
CWin-RM	9000	0	9000	2.000	58859.000	105.469	1040.423
Pmin	9000	0	9000	0.000	869.551	6.740	15.016

Sign test / Upper-tailed test:

N+	8203
Expected v	4150.000
Variance (N	2075.000
p-value (or	< 0.0001
alpha	0.05

The p-value is computed using an exact method.

Test interpretation: The average exeuction time of CWin-RM is larger than Pmin
H0: The two samples follow the same distribution.
Ha: The distributions of the two samples are different.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Wilcoxon signed-rank test / Upper-tailed test:

V	34290786
V (standarc	78.176
Expected v	17224575.000
Variance (V	47657528262.500
p-value (or	< 0.0001
alpha	0.05

The exact p-value could not be computed. An approximation has been used to compute the p-value.

Test interpretation:
H0: The two samples follow the same distribution.
Ha: The distributions of the two samples are different.
As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The risk to reject the null hypothesis H0 while it is true is lower than 0.01%.

Summary (p-values):

Variable\Tes	Sign test	on signed-rank test
FilteedRM	**< 0.0001**	**< 0.0001**

Figure A.5: A sign test and a Wilcoxon signed-rank test which suggest PMin is, on average, faster than CWin-RF.

List of Figures

List of Tables

List of Algorithms

Glossary

aperiodic task Aperiodic tasks are only characterized by a computation time and a deadline; there is no bound on the interarrival period of aperiodic tasks.

avionics Electronic systems used on aircraft, artificial satellites, and spacecraft. Avionic systems include communications, navigation, displays, autopilot, fuel management, etc.

certification Formally verify that a system conforms to standards and that its functional and non functional properties are respected, such as: the correct output is always produced under the timing restraints imposed for that operation.

certified A system formally verified that it conforms to standards and that its functional and nonfunctional properties are respected, such as the correct output is always produced under the timing restraints imposed for that operation.

composability Enforces that the behavior of individual components remain unchanged when considered in isolation and after their integration with all the other components constituting the systems.

compositionality Ensures that the overall application behavior can be provided by the composition of its constituting subcomponents.

criticality Critical level of the system, for example, high criticality refers to mission critical hardware, while low criticality means a fault of sorts does not endanger the system or put lives at risk.

deadline The latest time or date by which something should be completed.

deterministic A development model where no randomness is involved in the development of future states. A deterministic system will always produce the same output from a given starting condition or state.

extra-functional properties In a runtime monitoring context, everything that does not relate to the result produced or execution ordering, such as: a task A must complete within 10ms, power consumption must stay below 5W properties.

fault tolerance A guarantee of continuous operation even in the event a failure occurs.

feasible A task set is deemed feasible if, for all tasks, all jobs complete within timing and functional constraints.

functional properties In a runtime monitoring context, everything that relates to the produced result or execution ordering, such as: task A must execute before B; the sensor reading must be larger than 5.

hard real time A hard real-time system is a system that must operate within the confines of a stringent deadline. The application may incur catastrophic damage if it does not complete its function within the allotted time span.

harmonic task set A task set is deemed harmonic if the periods of its constituent tasks are all multiple of each other.

Integrated Modular Avionics Real time airborne computing systems architecture.

limited preemptive scheduler In limited- preemption, the scheduler is allowed to postpone a preemption based on some criteria.

loose harmonic task set A task set where task's periods are integer multiple of the smallest period in the system.

mission critical Hardware or software vital to the correct functioning of a system.

partition Virtualized computing environments that isolate system processes in space and time. By distributing processes among partitions, faults cannot propagate among partitions.

partitioning The process of dividing system resources into partitions so that they are completely isolated from one another. This isolation prevents faults from propagating to other partitions, improving the reliability, safety, and ease of development of the system.

periodic task Periodic tasks are characterized by a *constant* interarrival period, a computation time, and a deadline.

quality of service The overall performance of a computer system, particularly the performance seen by its users.

real time In this report real time relates to real time systems.

real-time systems A real-time system is one that must process information and produce a response within a specified time, else risk severe consequences, including failure.

response time analysis A response time analysis (RTA) calculates the maximum period of elapsed time from the release of a task until its computation is complete.

runtime The time during which an application is executing, in contrast to compile time, link time and load time.

safety-critical A system whose failure or malfunction may result in serious injury, death, or severe damage.

schedulability analysis A schedulability analysis determines if a given task set is feasible, that is if all tasks in the system complete within their assigned deadlines. A given schedulability analysis method may be sufficient or necessary. A sufficient schedulability test can only determine if a given task set is schedulable; if the test is false, then no conclusion can be drawn regarding the schedulability of the task set. A necessary schedulability analysis can determine if a task set is schedulable or unschedulable; it defines a property necessary to achieve schedulability, without whom, it is impossible for all tasks to meet their deadlines.

schedulable The evaluation, testing, and verification that the designated task set is always executed within a given time.

scheduling The Business Dictionary defines scheduling as determining when an activity should start or end, depending on its duration, predecessor activities and relationships, resource availability, and target completion date [27].

soft real time A soft real-time system is hardware or software that should operate at a given frequency. Failure to meet task deadlines can result in system degradation, and quality of service (QoS) can be affected but is otherwise without any consequences.

space partitioning A form of partitioning, space partitioning ensures that memory operations cannot cross partitions unless explicitly allowed to do so through shared memory or communication systems such as sockets.

sporadic task Sporadic tasks are characterized by a *minimum* interarrival period, a computation time, and a deadline.

static scheduler A static scheduling algorithm always assigns the same priority to a given task; Rate Monotonic (RM) is an example of a static fixed priority scheduler that assigns priority to tasks based on their period.

system jitter Delay variations caused by scheduling overhead. Jitter impact the execution times of tasks in the system.

task slack For a given task, slack is defined as the amount of time the task can remain idle until the ability to complete its computation within the allotted time limit is compromised. Slack time of a task can be impacted by the presence of other tasks in the system.

time partitioning A form of partitioning, time partitioning guarantees that the timing characteristics of tasks, such as their worst-case execution times are not affected by the execution of other tasks in other partitions.

unschedulable The task or task set fails to meet its time restrictions in at least one scenario.

work-conserving scheduler A work-conserving scheduler never allows the processor to be idle when the ready queue is not empty.

Acronyms

ACO Ant Colony Optimization.

API Application Program Interface.

B&B Branch and Bound.

C-EDF Clairvoyant EDF.

CISTER Research Center in Real-time and Embedded Computing Systems.

CPU Central Processing Unit.

CW-EDF Critical Time Window-Based Earliest Deadline First.

DM Deadline Monotonic.

DPS Deferred Preemption Scheduling.

EA Exact Algorithms.

EDF Earliest Deadline First.

EP-RM Efficient Precautious Rate Monotonic.

FCFS First-Come-First-Served.

FIFO First-In-First-Out.

FP Fixed Priority.

FPP Fixed Preemption Points.

GA Genetic Algorithms.

GNU GNU's Not Unix!.

Gr-EDF Group-Based EDF.

HA Heuristic Algorithms.

IIP Idle-Time Insertion Policy.

ILP Integer Linear Programming.

ILS Iterative Local Search.

IMA Integrated Modular Avionics.

LCM Least Common Multiple.

LIFO Last-In-First-Out.

LLF Least Laxity First.

LP-RM Lazy Precautious Rate Monotonic.

MC Model Checking.

MF Mathematical Formulation.

MH-A Meta-Heuristic Algorithms.

MILP Mixed Integer Linear Programming.

NP-EDF Non Preemptive Earliest Deadline First.

PN Petri Nets.

POSIX Portable Operating System Interface.

P-RM Precautious Rate Monotonic.

PTS Preemption Threshold Scheduling.

QoS Quality of Service.

RM Rate Monotonic.

RTA Response Time Analysis.

SA Simulated Annealing.

SAT satisfiability module theories.

SJF Shortest Job First.

TS Tabu Search.

WCET Worst Case Execution Time.

References

[1] T. F. Abdelzaher and K. G. Shin. "Optimal combined task and message scheduling in distributed real-time systems". In: *Proceedings 16th IEEE Real-Time Systems Symposium.* 1995, pp. 162–171. DOI: `10.1109/REAL.1995.495206`.

[2] Federal Aviation Administration. *Aeronautical Information Manual.* `https://www.faa.gov/air_traffic/publications/media/aim.pdf`. 2015.

[3] Federal Aviation Administration. *Air Carrier Operational Approval and Use of TCAS II.* `https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/903609`. 2013.

[4] Federal Aviation Administration. *Introduction to TCAS II.* `https://www.faa.gov/documentLibrary/media/Advisory_Circular/TCAS%20II%20V7.1%20Intro%20booklet.pdf`. 2011.

[5] Airlines electronic engineering committee (AEEC). *Avionics Application Software Standard Interface specification 653-2.* 2006.

[6] Radio Technical Commission for Aeronautics. *Design Assurance Guidance for Airborne Electronic Hardware.* 2000.

[7] Radio Technical Commission for Aeronautics. *DO-178C Software Considerations in Airborne Systems and Equipment Certification.* 2011.

[8] Radio Technical Commission for Aeronautics. *Software Considerations in Airborne Systems and Equipment Certification.* 1992.

[9] Vincent Poirriez; Nicola Yanev; Rumen Andonov. "A hybrid algorithm for the unbounded knapsack problem". In: *Discrete Optimization* 6 (1 2009). DOI: `10.1016/j.disopt.2008.09.004`.

[10] N.C. Audsley. "[IEE IEE Colloquium on Hybrid Control for Real-Time Systems - London, UK (6 Dec. 1996)] IEE Colloquium on Hybrid Control for Real-Time Systems - Flexible scheduling theory for advanced engine controllers". In: vol. 1996. 1996. DOI: `10.1049/ic:19961370`.

[11] T. P. Baker and A. Shaw. "The cyclic executive model and Ada". In: *Proceedings. Real-Time Systems Symposium.* 1988, pp. 120–129. DOI: `10.1109/REAL.1988.51108`.

[12] S.; Maciel P. Barreto R.; Cavalcante. "[IEEE 24th International Conference on Distributed Computing Systems Workshops, 2004. Proceedings. - Hachioji, Tokyo, Japan (2004.03.24-2004.03.24)] 24th International Conference on Distributed Computing Systems Workshops, 2004. Proceedings. - A time Petri net approach for finding preruntime schedules in embedded hard real-time systems". In: 2004. ISBN: 0-7695-2087-1. DOI: `10.1109/icdcsw.2004.1284131`.

[13] Sanjoy Baruah. "The limited-preemption uniprocessor scheduling of sporadic task systems". In: *17th Euromicro Conference on Real-Time Systems (ECRTS'05).* 2005, pp. 137–144. DOI: `10.1109/ECRTS.2005.32`.

[14] Leonora Bianchi et al. "A survey on metaheuristics for stochastic combinatorial optimization". In: *Natural Computing* 8.2 (2009), pp. 239–287. ISSN: 1572-9796. DOI: `10.1007/s11047-008-9098-4`. URL: `http://dx.doi.org/10.1007/s11047-008-9098-4`.

[15] Enrico Bini and Giorgio C. Buttazzo. "Measuring the Performance of Schedulability Tests". In: *Real-Time Systems* 30.1 (2005), pp. 129–154. ISSN: 1573-1383. DOI: `10.1007/s11241-005-0507-9`. URL: `http://dx.doi.org/10.1007/s11241-005-0507-9`.

[16] Christian Blum and Andrea Roli. “Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison”. In: *ACM Comput. Surv.* 35.3 (Sept. 2003), pp. 268–308. ISSN: 0360-0300. DOI: `10.1145/937503.937505`. URL: `http://doi.acm.org/10.1145/937503.937505`.

[17] Boeing. *Certifying Boeing’s Airplanes.* `http://787updates.newairplane.com/Certification-Process`. 2015.

[18] Bach D Bui et al. “Impact of cache partitioning on multi-tasking real time embedded systems”. In: *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA’08. 14th IEEE International Conference on.* IEEE. 2008, pp. 101–110.

[19] A. Burns, N. Hayes, and M.F. Richardson. “Generating Feasible Cyclic Schedules”. In: *Control Engineering Practice* (1995), pp. 151–162.

[20] A. Burns and J. A. McDermid. “Real-time safety-critical systems: analysis and synthesis”. In: *Software Engineering Journal* 9.6 (1994), pp. 267–281. ISSN: 0268-6961. DOI: `10.1049/sej.1994.0036`.

[21] Alan Burns. “Advances in Real-time Systems”. In: ed. by Sang H. Son. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995. Chap. Preemptive Priority-based Scheduling: An Appropriate Engineering Approach, pp. 225–248. ISBN: 0-13-083348-7. URL: `http://dl.acm.org/citation.cfm?id=207721.207731`.

[22] G. C. Buttazzo, M. Bertogna, and G. Yao. “Limited Preemptive Scheduling for Real-Time Systems. A Survey”. In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 3–15. ISSN: 1551-3203. DOI: `10.1109/TII.2012.2188805`.

[23] Hua Chen and Albert M. K. Cheng. “Applying Ant Colony Optimization to the Partitioned Scheduling Problem for Heterogeneous Multiprocessors”. In: *SIGBED Rev.* 2.2 (Apr. 2005), pp. 11–14. ISSN: 1551-3688. DOI: `10.1145/1121788.1121793`. URL: `http://doi.acm.org/10.1145/1121788.1121793`.

[24] Barry Chiff. *Proficient pilot automation dependency.* `https://www.aopa.org/news-and-media/all-news/2013/september/01/proficient-pilot-automation-dependency`. 2013.

[25] Stephen A. Cook. “The Complexity of Theorem-proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing.* STOC ’71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: `10.1145/800157.805047`. URL: `http://doi.acm.org/10.1145/800157.805047`.

[26] Nikolaj De Moura Leonardo; Bjørner. “Satisfiability modulo theories : introduction and applications”. In: *Communications of the ACM* 54 (9 2011). DOI: `10.1145/1995376.1995394`.

[27] Business Dictionary. *Definition of Scheduling.* `http://www.businessdictionary.com/definition/scheduling.html`. 2015.

[28] M. DiNatale and J. A. Stankovic. “Applicability of simulated annealing methods to real-time scheduling and jitter control”. In: *Proceedings 16th IEEE Real-Time Systems Symposium.* 1995, pp. 190–199. DOI: `10.1109/REAL.1995.495209`.

[29] Marco Dorigo. “Optimization, learning and natural algorithms”. In: *Ph. D. Thesis, Politecnico di Milano, Italy* (1992).

[30] C. Ekelin. “Clairvoyant non-preemptive EDF scheduling”. In: *18th Euromicro Conference on Real-Time Systems (ECRTS’06).* 2006, 7 pp.–32. DOI: `10.1109/ECRTS.2006.7`.

[31] international Electrotechnical Commission. *Functional safety of electrical/electronic/programmable electronic safety-related systems.* 2010.

[32] Alexandre Esper et al. “How Realistic is the Mixed-criticality Real-time System Model?” In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems.* RTNS ’15. Lille, France: ACM, 2015, pp. 139–148. ISBN: 978-1-4503-3591-1. DOI: `10.1145/2834848.2834869`. URL: `http://doi.acm.org/10.1145/2834848.2834869`.

[33] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[34] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[35] Laurent George, Nicolas Rivierre, and Marco Spuri. *Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling.* Research Report RR-2966. Projet REFLECS. INRIA, 1996. URL: `https://hal.inria.fr/inria-00073732`.

[36] Fred Glover. "Tabu Search—Part I". In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206. DOI: `10.1287/ijoc.1.3.190`. eprint: `http://dx.doi.org/10.1287/ijoc.1.3.190`. URL: `http://dx.doi.org/10.1287/ijoc.1.3.190`.

[37] E. Grolleau and A. Choquet-Geniet. "Off-Line Computation of Real-Time Schedules Using Petri Nets". In: *Discrete Event Dynamic Systems* 12.3 (July 2002), pp. 311–333. ISSN: 0924-6703. DOI: `10.1023/A:1015673516542`. URL: `http://dx.doi.org/10.1023/A:1015673516542`.

[38] Nan Guan and Wang Yi. *Fixed-Priority Multiprocessor Scheduling: Critical Instant, Response Time and Utilization Bound.* 2012.

[39] Nan Guan and Wang Yi. "General and Efficient Response Time Analysis for EDF Scheduling". In: *Proceedings of the Conference on Design, Automation & Test in Europe.* DATE '14. Dresden, Germany: European Design and Automation Association, 2014, 255:1–255:6. ISBN: 978-3-9815370-2-4. URL: `http://dl.acm.org/citation.cfm?id=2616606.2616986`.

[40] B. Hamidzadeh and D. J. Lilja. "Dynamic scheduling strategies for shared-memory multiprocessors". In: *Proceedings of 16th International Conference on Distributed Computing Systems.* 1996, pp. 208–215. DOI: `10.1109/ICDCS.1996.507918`.

[41] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: `10.1109/TSSC.1968.300136`.

[42] Vance Hilderman and Len Buckwalter. *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware).* 1st. USA: Avionics Communications Inc., 2007. ISBN: 1885544251, 9781885544254.

[43] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA: MIT Press, 1992. ISBN: 0262082136.

[44] Sartaj Horowitz Ellis; Sahni. "Computing Partitions with Applications to the Knapsack Problem". In: *Journal of the ACM* 21 (2 Apr. 1974). DOI: `10.1145/321812.321823`.

[45] Wind River Intel. *VxWORKS.* `"https://www.windriver.com/products/vxworks/"`.

[46] Jean-Bernard Itier. "A380 integrated modular avionics". In: *Proceedings of the ARTIST2 meeting on integrated modular avionics.* Vol. 1. 2. 2007, pp. 72–75.

[47] K. Jeffay, D. F. Stanat, and C. U. Martel. "On non-preemptive scheduling of period and sporadic tasks". In: *[1991] Proceedings Twelfth Real-Time Systems Symposium.* 1991, pp. 129–139. DOI: `10.1109/REAL.1991.160366`.

[48] Kevin Jeffay, Donald F Stanat, and Charles U Martel. "On non-preemptive scheduling of period and sporadic tasks". In: *Real-Time Systems Symposium, 1991. Proceedings., Twelfth.* IEEE. 1991, pp. 129–139.

[49] Jan Jonsson and Kang G Shin. "A parametrized branch-and-bound strategy for scheduling precedence-constrained tasks on a multiprocessor system". In: *Parallel Processing, 1997., Proceedings of the 1997 International Conference on.* IEEE. 1997, pp. 158–165.

[50] JUnit. *JUnit.* `"http://junit.org/junit4/"`. 2017.

[51] U. Keskin, R. J. Bril, and J. J. Lukkien. "Exact response-time analysis for fixed-priority preemption-threshold scheduling". In: *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010).* 2010, pp. 1–4. DOI: `10.1109/ETFA.2010.5640984`.

[52] Y. Laalaoui et al. "Ant colony system with stagnation avoidance for the scheduling of real-time tasks". In: *2009 IEEE Symposium on Computational Intelligence in Scheduling.* 2009, pp. 1–6. DOI: `10.1109/SCIS.2009.4927007`.

[53] Yacine Laalaoui and Nizar Bouguila. “Review: Pre-run-time Scheduling in Real-time Systems: Current Researches and Artificial Intelligence Perspectives”. In: *Expert Syst. Appl.* 41.5 (Apr. 2014), pp. 2196–2210. ISSN: 0957-4174. DOI: `10.1016/j.eswa.2013.09.018`. URL: `http://dx.doi.org/10.1016/j.eswa.2013.09.018`.

[54] Peter J. M. van Laarhoven and Emile H. L. Aarts. “Simulated annealing”. In: *Simulated Annealing: Theory and Applications*. Dordrecht: Springer Netherlands, 1987, pp. 7–15. ISBN: 978-94-015-7744-1. DOI: `10.1007/978-94-015-7744-1_2`. URL: `http://dx.doi.org/10.1007/978-94-015-7744-1_2`.

[55] A. H. Land and A. G. Doig. “An automatic method for solving discrete programming problems”. In: *ECONOMETRICA* 28.3 (1960), pp. 497–520.

[56] Ailsa H Land and Alison G Doig. “An automatic method of solving discrete programming problems”. In: *Econometrica: Journal of the Econometric Society* (1960), pp. 497–520.

[57] Seppo J. Laplante Phillip A.; Ovaska. “Real-Time Systems Design and Analysis (Tools for the Practitioner) || Software Design Approaches”. In: 2011. ISBN: 9781118136607,9780470768648. DOI: `10.1002/9781118136607.ch6`.

[58] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN: 0131489062.

[59] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201695812.

[60] D.; Younis M.; Zhou J. Lee Y.-H.; Kim. “[IEEE 19th Digital Avionics System Conference. Proceedings - Philadelphia, PA, USA (7-13 Oct. 2000)] 19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No.00CH37126) - Scheduling tool and algorithm for integrated modular avionics systems”. In: vol. 1. 2000. ISBN: 0-7803-6395-7. DOI: `10.1109/DASC.2000.886885`.

[61] Suk Kyoon Lee. “On-line multiprocessor scheduling algorithms for real-time tasks”. In: *TENCON ’94. IEEE Region 10’s Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*. 1994, 607–611 vol.2. DOI: `10.1109/TENCON.1994.369148`.

[62] Wenming Li, Krishna Kavi, and Robert Akl. “A Non-preemptive Scheduling Algorithm for Soft Real-time Systems”. In: *Comput. Electr. Eng.* 33.1 (Jan. 2007), pp. 12–29. ISSN: 0045-7906. DOI: `10.1016/j.compeleceng.2006.04.002`. URL: `http://dx.doi.org/10.1016/j.compeleceng.2006.04.002`.

[63] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: `10.1145/321738.321743`. URL: `http://doi.acm.org/10.1145/321738.321743`.

[64] Leonardo Mangeruca et al. “Uniprocessor Scheduling Under Precedence Constraints for Embedded Systems Design”. In: *ACM Trans. Embed. Comput. Syst.* 7.1 (Dec. 2007), 6:1–6:30. ISSN: 1539-9087. DOI: `10.1145/1324969.1324975`. URL: `http://doi.acm.org/10.1145/1324969.1324975`.

[65] Robert C. Martin. *Design Principles and Design Patterns*. 2000.

[66] G. B. Mathews. “On the Partition of Numbers”. In: *Proceedings of the London Mathematical Society* s1-28 (1 Nov. 1896). DOI: `10.1112/plms/s1-28.1.486`.

[67] A. Metzner et al. “Scheduling distributed real-time systems by satisfiability checking”. In: *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA’05)*. 2005, pp. 409–415. DOI: `10.1109/RTCSA.2005.90`.

[68] J. Michael Moore. “An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs”. In: *Management Science* 15.1 (1968), pp. 102–109. DOI: `10.1287/mnsc.15.1.102`. URL: `https://doi.org/10.1287/mnsc.15.1.102`.

[69] M. Nasri and B. B. Brandenburg. “Offline Equivalence: A Non-preemptive Scheduling Technique for Resource-Constrained Embedded Real-Time Systems (Outstanding Paper)”. In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2017, pp. 75–86. DOI: `10.1109/RTAS.2017.34`.

[70] M. Nasri and G. Fohler. "Non-work-conserving Non-preemptive Scheduling: Motivations, Challenges, and Potential Solutions". In: *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 2016, pp. 165–175. DOI: 10.1109/ECRTS.2016.11.

[71] Mitra Nasri and Gerhard Fohler. "Non-work-conserving Scheduling of Non-preemptive Hard Real-time Tasks Based on Fixed Priorities". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. RTNS '15. Lille, France: ACM, 2015, pp. 309–318. ISBN: 978-1-4503-3591-1. DOI: 10.1145/2834848.2834856. URL: http://doi.acm.org/10.1145/2834848.2834856.

[72] Mitra Nasri and Mehdi Kargahi. "Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks". In: *Real-Time Systems* 50.4 (2014), pp. 548–584. ISSN: 1573-1383. DOI: 10.1007/s11241-014-9203-y. URL: http://dx.doi.org/10.1007/s11241-014-9203-y.

[73] Mitra Nasri et al. "On the Optimality of RM and EDF for Non-Preemptive Real-Time Harmonic Tasks". In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versaille, France: ACM, 2014, 331:331–331:340. ISBN: 978-1-4503-2727-5. DOI: 10.1145/2659787.2659806. URL: http://doi.acm.org/10.1145/2659787.2659806.

[74] Dominik Niedermeie and Anthony A. Lambregts. *Fly-By-Wire Augmented Manual Control - Basic Design Considerations*. http://www.icas.org/ICAS_ARCHIVE/ICAS2012/PAPERS/605.PDF. 2012.

[75] Roman Nossal. "An Evolutionary Approach to Multiprocessor Scheduling of Dependent Tasks". In: *Future Gener. Comput. Syst.* 14.5-6 (Dec. 1998), pp. 383–392. ISSN: 0167-739X. DOI: 10.1016/S0167-739X(98)00041-7. URL: http://dx.doi.org/10.1016/S0167-739X(98)00041-7.

[76] Sung-Heun Oh and Seung-Min Yang. "A Modified Least-Laxity-First scheduling algorithm for real-time tasks". In: *Proceedings Fifth International Conference on Real-Time Computing Systems and Applications (Cat. No.98EX236)*. 1998, pp. 31–36. DOI: 10.1109/RTCSA.1998.726348.

[77] Carl Adam Petri. "Kommunikation mit automaten". In: (1962).

[78] David Pisinger. "Where are the hard knapsack problems". In: *Computers and Operations Research* 32 (9 2005). DOI: 10.1016/j.cor.2004.03.002.

[79] Michael J Pont et al. "Selecting an appropriate scheduler for use with time-triggered embedded systems." In: *EuroPLoP*. 2007, pp. 595–618.

[80] Paul Pop et al. "Design Optimization of Multi-Cluster Embedded Systems for Real-Time Applications". In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 21028–. ISBN: 0-7695-2085-5. URL: http://dl.acm.org/citation.cfm?id=968879.969152.

[81] Paul J. Prisaznuk. "[IEEE 2008 IEEE/AIAA 27th Digital Avionics Systems Conference (DASC) - St. Paul, MN, USA (2008.10.26-2008.10.30)] 2008 IEEE/AIAA 27th Digital Avionics Systems Conference - ARINC 653 role in Integrated Modular Avionics (IMA)". In: 2008. ISBN: 978-1-4244-2207-4. DOI: 10.1109/DASC.2008.4702770.

[82] Eric; Pagetti Claire Puffitsch Wolfgang; Noulard. "Off-line mapping of multi-rate dependent task sets to many-core platforms". In: *Real-Time Systems* (July 2015). DOI: 10.1007/s11241-015-9232-1.

[83] William Pugh. "Skip lists: A probabilistic alternative to balanced trees". In: *Algorithms and Data Structures: Workshop WADS '89 Ottawa, Canada, August 17–19, 1989 Proceedings*. Ed. by F. Dehne, J. R. Sack, and N. Santoro. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 437–449. ISBN: 978-3-540-48237-6. DOI: 10.1007/3-540-51542-9_36. URL: https://doi.org/10.1007/3-540-51542-9_36.

[84] R. Andonov; V. Poirriez; S. Rajopadhye. "Unbounded knapsack problem: Dynamic programming revisited". In: *European Journal of Operational Research* 123 (2 2000). DOI: 10.1016/s0377-2217(99)00265-9.

[85] Anders P. Ravn and Martin Schoeberl. “Cyclic Executive for Safety-critical Java on Chip-multiprocessors”. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES ’10. Prague, Czech Republic: ACM, 2010, pp. 63–69. ISBN: 978-1-4503-0122-0. DOI: `10.1145/1850771.1850779`. URL: `http://doi.acm.org/10.1145/1850771.1850779`.

[86] *Real-Time Design Patterns-Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Professional. ISBN: 0-201-69956-7.

[87] “Real-time systems: Design principles for distributed embedded applications: By Hermann Kopetz. Kluwer Academic Publishers, Boston, MA. (1997).” In: *Computers & Mathematics with Applications* 34 (10 1997). DOI: `10.1016/s0898-1221(97)90277-7`.

[88] Philip E. Ross. *When Will We Have Unmanned Commercial Airliners?* `http://spectrum.ieee.org/aerospace/aviation/when-will-we-have-unmanned-commercial-airliners`. 2011.

[89] Manas Saksena and Yun Wang. “Scalable Real-time System Design Using Preemption Thresholds”. In: *Proceedings of the 21st IEEE Conference on Real-time Systems Symposium*. RTSS’10. Orlando, Florida: IEEE Computer Society, 2000, pp. 25–34. ISBN: 0-7695-0900-2. URL: `http://dl.acm.org/citation.cfm?id=1890629.1890633`.

[90] Scala. *Scala*. `"https://www.scala-lang.org/"`. 2017.

[91] *Scalameter*. `https://scalameter.github.io/`.

[92] L. Sha, R. Rajkumar, and J. P. Lehoczky. “Priority inheritance protocols: an approach to real-time synchronization”. In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. ISSN: 0018-9340. DOI: `10.1109/12.57058`.

[93] Lui Sha, Mark H. Klein, and John B. Goodenough. “Rate Monotonic Analysis for Real-Time Systems”. In: *Foundations of Real-Time Computing: Scheduling and Resource Management*. Ed. by André M. van Tilborg and Gary M. Koob. Boston, MA: Springer US, 1991, pp. 129–155. ISBN: 978-1-4615-3956-8. DOI: `10.1007/978-1-4615-3956-8_5`. URL: `http://dx.doi.org/10.1007/978-1-4615-3956-8_5`.

[94] C. Shen, K. Ramamritham, and J. A. Stankovic. *Resource Reclaiming in Real-Time*. Tech. rep. Amherst, MA, USA, 1990.

[95] M. Short. “Improved schedulability analysis of implicit deadline tasks under limited preemption EDF scheduling”. In: *ETFA2011*. 2011, pp. 1–8. DOI: `10.1109/ETFA.2011.6059008`.

[96] Skybrary. *Autopilot*. `http://www.skybrary.aero/index.php/Autopilot`. 2017.

[97] Skybrary. *Cockpit Automation - Advantages and Safety Challenges*. `http://www.skybrary.aero/index.php/Cockpit_Automation_-_Advantages_and_Safety_Challenges`. 2017.

[98] Marco Spuri. *Analysis of Deadline Scheduled Real-Time Systems*. Research Report RR-2772. Projet REFLECS. INRIA, 1996. URL: `https://hal.inria.fr/inria-00073920`.

[99] Roger Stafford. *Random Vectors with Fixed Sum*. `http://www.mathworks.com/matlabcentral/fileexchange/9700`. 2006.

[100] International Organization for Standardization (ISO). *Road vehicles – Functional safety*. 2011.

[101] John A. Stankovic. “Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems”. In: *Computer* 21.10 (Oct. 1988), pp. 10–19. ISSN: 0018-9162. DOI: `10.1109/2.7053`. URL: `http://dx.doi.org/10.1109/2.7053`.

[102] SYSGO. *PikeOs*. `"https://www.sysgo.com/products/pikeos-hypervisor/"`.

[103] D. Tamas-Selicean and P. Pop. “Optimization of Time-Partitions for Mixed-Criticality Real-Time Distributed Embedded Systems”. In: *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. 2011, pp. 1–10. DOI: `10.1109/ISORCW.2011.11`.

[104] Lynx Software Technologies. *LynxOS-178*. `"http://www.lynx.com/products/real-time-operating-systems/lynxos-178-rtos-for-do-178b-software-certification/"`.

[105] Project MAC (Massachusetts Institute of Technology). Engineering Robotics Group and ML Dertouzos. *Control robotics: The procedural control of physical processes*. 1973.

[106] Bureau of Transportation Statistics. *Airline Fuel Cost and Consumption (U.S. Carriers - Scheduled).* `https://www.transtats.bts.gov/fuel.asp`. 2016.

[107] Domiţian Tămaş-Selicean and Paul Pop. "Design Optimization of Mixed-Criticality Real-Time Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 14.3 (Apr. 2015), 50:1–50:29. ISSN: 1539-9087. DOI: `10.1145/2700103`. URL: `http://doi.acm.org/10.1145/2700103`.

[108] J.D. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384 –393. ISSN: 0022-0000. DOI: `http://dx.doi.org/10.1016/S0022-0000(75)80008-0`. URL: `http://www.sciencedirect.com/science/article/pii/S0022000075800080`.

[109] S. Voss and B. Schätz. "Deployment and Scheduling Synthesis for Mixed-Critical Shared-Memory Applications". In: *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS).* 2013, pp. 100–109. DOI: `10.1109/ECBS.2013.23`.

[110] Yun Wang and Manas Saksena. "Scheduling Fixed-Priority Tasks with Preemption Threshold". In: *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications.* RTCSA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 328–. ISBN: 0-7695-0306-3. URL: `http://dl.acm.org/citation.cfm?id=519167.828730`.

[111] K. W. Tindell; A. Burns; A. J. Wellings. "Allocating hard real-time tasks: An NP-Hard problem made easy". In: *Real-Time Systems* 4 (2 June 1992). DOI: `10.1007/bf00365407`.

[112] J. Xu. "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations". In: *IEEE Transactions on Software Engineering* 19.2 (1993), pp. 139–154. ISSN: 0098-5589. DOI: `10.1109/32.214831`.

[113] Jia Xu and David Parnas. "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations". In: *IEEE Trans. Softw. Eng.* 16.3 (Mar. 1990), pp. 360–369. ISSN: 0098-5589. DOI: `10.1109/32.48943`. URL: `http://dx.doi.org/10.1109/32.48943`.

[114] Jia Xu and David Lorge Parnas. "Priority Scheduling Versus Pre-Run-Time Scheduling". In: *Real-Time Syst.* 18.1 (Jan. 2000), pp. 7–23. ISSN: 0922-6443. DOI: `10.1023/A:1008198310125`. URL: `http://dx.doi.org/10.1023/A:1008198310125`.

[115] Yun Liang Tulika Mitra Yan Li Vivy Suhendra and Abhik Roychoudhury. "Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores". In: *Real-Time Systems Symposium* (2009).

[116] G. Yao, G. Buttazzo, and M. Bertogna. "Comparative evaluation of limited preemptive methods". In: *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010).* 2010, pp. 1–8. DOI: `10.1109/ETFA.2010.5641199`.

[117] F. Zhang and A. Burns. "Schedulability Analysis for Real-Time Systems with EDF Scheduling". In: *IEEE Transactions on Computers* 58.9 (2009), pp. 1250–1258. ISSN: 0018-9340. DOI: `10.1109/TC.2009.58`.