Mälardalen University Press Dissertations
No. 79

A STUDY OF COMBINATORIAL OPTIMIZATION PROBLEMS IN INDUSTRIAL COMPUTER SYSTEMS

Markus Bohlin

2009

School of Innovation, Design and Engineering

ISSN 1651-4238
ISBN 978-91-86135-47-8
Printed by Mälardalen University, Västerås, Sweden

Mälardalen University Press Dissertations
No. 79

A STUDY OF COMBINATORIAL OPTIMIZATION PROBLEMS IN INDUSTRIAL COMPUTER SYSTEMS

Markus Bohlin

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik, kommer att offentligen försvaras måndagen 14 december, 2009, 14.00 i Beta, Mälardalens högskola, Västerås.

Fakultetsopponent: prof. Petru Eles, Linköpings Universitet

Akademin för innovation, design och teknik

Abstract

A combinatorial optimization problem is an optimization problem where the number of possible solutions is finite and grows combinatorially with the problem size. Combinatorial problems exist everywhere in industrial systems. This thesis focuses on solving three such problems which arise within two different areas where industrial computer systems are often used. Within embedded systems and real-time systems, we investigate the problems of allocating stack memory for a system where a shared stack may be used, and of estimating the highest response time of a task in a system of industrial complexity. We propose a number of different algorithms to compute safe upper bounds on run-time stack usage whenever the system supports stack sharing. The algorithms have in common that they can exploit commonly-available information regarding timing behavior of the tasks in the system. Given upper bounds on the individual stack usage of the tasks, it is possible to estimate the worst-case stack behavior by analyzing the possible and impossible preemption patterns. Using relations on offset and precedences, we form a preemption graph, which is further analyzed to find safe upper-bounds on the maximal preemptions chain in the system. For the special case where all tasks exist in a single static schedule and share a single stack, we propose a polynomial algorithm to solve the problem. For generalizations of this problem, we propose an exact branch-and-bound algorithm for smaller problems and a polynomial heuristic algorithm for cases where the branch-and-bound algorithm fails to find a solution in reasonable time. All algorithms are evaluated in comprehensive experimental studies. The polynomial algorithm is implemented and shipped in the developer tool set for a commercial real-time operating system, Rubus OS. The second problem we study in the thesis is how to estimate the highest response time of a specified task in a complex industrial real-time system. The response-time analysis is done using a best-effort approach, where a detailed model of the system is simulated on input constructed using a local search procedure. In an evaluation on three different systems we can see that the new algorithm was able to produce higher response times much faster than what has previously been possible. Since the analysis is based on simulation and measurement, the results are not safe in the sense that they are always higher or equal to the true response time of the system. The value of the method lies instead in that it makes it possible to analyze complex industrial systems which cannot be analyzed accurately using existing safe approaches. The third problem is in the area of maintenance planning, and focus on how to dynamically plan maintenance for industrial systems. Within this area we have focused on industrial gas turbines and rail vehicles. We have developed algorithms and a planning tool which can be used to plan maintenance for gas turbines and other stationary machinery. In such problems, it is often the case that performing several maintenance actions at the same time is beneficial, since many of these jobs can be done in parallel, which reduces the total downtime of the unit. The core of the problem is therefore how to (or how not to) group maintenance activities so that a composite cost due to spare parts, labor and loss of production due to downtime is minimized. We allow each machine to have individual schedules for each component in the system. For rail vehicles, we have evaluated the effect of re-planning maintenance in the case where the component maintenance deadline is set to reflect a maximum risk of breakdown in a Gaussian failure distribution. In such a model, we show by simulation that re-planning of maintenance can reduce the number of maintenance stops when the variance and expected value of the distribution are increased. For the gas turbine maintenance planning problem, we have evaluated the planning software on a real-world scenario from the oil and gas industry and compared it to the solutions obtained from a commercial integer programming solver. It is estimated that the availability increase from using our planning software is between 0.5 to 1.0 %, which is substantial considering that availability is currently already at 97-98 %.

ISSN 1651-4238
ISBN 978-91-86135-47-8


Swedish
Institute of
Computer
Science
SICS

Abstract

A combinatorial optimization problem is an optimization problem where the number of possible solutions is finite and grows combinatorially with the problem size. Combinatorial problems exist everywhere in industry. This thesis focuses on solving three such problems which arise within two different areas where industrial computer systems are often used. Within embedded and real-time systems, we investigate the problems of allocating stack memory for a system where a shared stack may be used, and of estimating the highest response time of a task in a system of industrial complexity. We propose a number of different algorithms to compute safe upper bounds on run-time stack usage whenever the system supports stack sharing. The algorithms have in common that they can exploit commonly-available information regarding timing behavior of the tasks in the system. Given upper bounds on the individual stack usage of the tasks, it is possible to estimate the worst-case stack behavior by analyzing the possible and impossible preemption patterns. Using relations on offset and precedences, we form a preemption graph, which is further analyzed to find safe upper-bounds on the maximal preemptions chain in the system. For the special case where all tasks exist in a single static schedule and share a single stack, we propose a polynomial algorithm to solve the problem. For generalizations of this problem, we propose an exact branch-and-bound algorithm for smaller problems and a polynomial heuristic algorithm for cases where the branch-and-bound algorithm fails to find a solution in reasonable time. All algorithms are evaluated in comprehensive experimental studies. The polynomial algorithm is implemented and shipped in the developer tool set for a commercial real-time operating system, Rubus OS.

The second problem we study in the thesis is how to estimate the highest response time of a specified task in a complex industrial real-time system. The response-time analysis is done using a best-effort approach, where a detailed model of the system is simulated on input constructed using a local search pro-

cedure. In an evaluation on three different models we can see that the new algorithm was able to produce higher response times much faster than a previous approach based on an evolutionary algorithm. Since the analysis is based on simulation and measurement, the results are not safe in the sense that they are not always higher or equal to the true response time. The value of the method lies instead in that it makes it possible to analyze complex industrial systems which cannot be analyzed accurately using existing safe approaches.

The third problem is in the area of maintenance planning, and focuses on how to dynamically plan maintenance for industrial systems. Within this area we have focused on industrial gas turbines and rail vehicles. We have developed algorithms and a planning tool which can be used to plan maintenance for gas turbines and other stationary machinery. In such problems, it is often the case that performing several maintenance actions at the same time is beneficial, since many of these jobs can be done in parallel, which reduces the total downtime of the unit. The core of the problem is therefore how to (or how not to) group maintenance activities so that a composite cost due to spare parts, labor and loss of production due to downtime can be minimized. We allow each machine to have individual schedules for each component in the system. For rail vehicles, we have evaluated the effect of re-planning maintenance in the case where the component maintenance deadline is set to reflect the maximum tolerable risk of subsystem usage counter overrun, modeled using a Gaussian distribution. In such a model, we show by simulation that re-planning of maintenance can reduce the number of maintenance stops when the variance and expected value of the distribution are increased. For the gas turbine maintenance planning problem, we have evaluated the planning software on a real-world scenario from the oil and gas industry and compared it to the solutions obtained from a commercial integer programming solver. It is estimated that the availability increase from using our planning software is between 0.5 to 1.0 %, which is substantial considering that availability is currently already at 97–98 %.

To my family

Acknowledgments

During my doctoral studies I have had the pleasure to work with many talented, intelligent people. In this limited space I will try my best to express my gratitude to you. First of all I would like to thank my main advisor Björn Lisper, my industrial advisor Per Kreuger and my co-advisor Mikael Sjödin. You have been a great help with a ton of issues and have also complemented each other surprisingly well: Björn have kept his eyes on the details and the process, which have helped in getting things right, Per have guided me past obstacles in the combinatorial optimization area, as well as forced me to focus on other things than local search (or some other method I have preferred over the years), and Mikael deserves much credit for being relentless in the quest for the perfect scientific paper, thereby motivating me to work harder.

My time as a doctoral student has been particularly varied since I've also worked at the Swedish Institute of Computer Science (SICS). At SICS, Martin Aronsson has been my informal mentor, for which I am very grateful. I hope we can continue working together, since there always are new things I can learn from you. Björn Levin: you deserve my deepest gratitude for securing funding during my studies, as well as having supported me in my ambitions in plenty of other areas. Your leadership skills and willpower have continued to impress me; the latter has also given me ample opportunity to practice my argumentation skills! Anders Holst, Jan Ekman, Malin Forsgren, Rebecca Steinert, Kivanc Doganay and Javier Ubillos: you have all become much more than colleagues in the course of my work at SICS, and I'm very much looking forward to continue working with you. Other people I have particularly enjoyed working with include Charlotta Jörsäter, Janusz Launberg and Tobias Bexelius.

Being a research scientist at SICS, a natural part of work has been to collect information and disseminate and deploy results in industry. I have therefore had the great fortune of being able to work with people from the "real world",

discussing and trying to solve real problems that actually matters in industry. At Siemens Industrial Turbomachinery AB I have met many competent and open individuals, not being afraid to speak their mind and always striving for excellence. Of those, two people in particular deserve my gratitude. Mathias Wärja, the manager of the project I had the pleasure of participating in, is of the most inspiring and energetic persons I have ever met. Pontus Slottner has taught me a lot about how a gas turbine works and in particular how it deteriorates in different conditions. I've really enjoyed working, travelling and spending time with you guys. I also want to thank Bengt Svensson for his time and patience with a newcomer in the field, Mathias Persson for his help regarding planning of gas turbine maintenance activities, and Sven-Gunnar Sundkvist for our pleasant conversations about the United States in general and the Grand Canyon in particular.

I have also worked with several people at Bombardier Transportation in Västerås. First of all I want to thank Ulf Westberg for his critical eyes and involvement in our shared research projects. I'm very much looking forward to working with you in the future. I also want to thank Peter Oom and Ola Sellin for their patience and support. Finally, Stefan Larsen has been a great support and showed much enthusiasm over the years, which has helped tremendously.

Doing research at Mälardalen University (MdH) has been a quite different experience from working at SICS. Being a research university, the focus have always been on producing excellent research and publications, but the close ties that MdH has with industry also helps in keeping the research focussed on real-world problems with industrial and practical relevance. But what I have enjoyed the most is spending time with the people there. Working with my close friend and colleague Jan Carlson has always been inspirational and great fun. I have lost track of the number of small and large favors you have granted me, but I sincerely hope that I can make it up to you some day. Other people I have been working with and that deserve my deepest gratitude include Kaj Hänninen, Johan Kraft, Jukka Mäki-Turja, Yue Lu, Thomas Nolte, Radu Dobrin, and Waldemar Kocjan.

Finally, for being the sunshine in my life, I am eternally grateful to my wife Veronica and my daughter Idun.

Thank you!

Pasadena, CA, USA
July 2009

Contents

List of Figures

List of Tables

List of Algorithms

I

Thesis

Chapter 1

Introduction

As long as man has existed, he has tried to do his best, given what he possesses and the current circumstances affecting him. In the modern day (and in a more formal setting), this activity is called *optimization*, and is normally undertaken with the goal of minimizing or maximizing some form of *objective function*. Optimization is an activity whose importance cannot be overstated, and its presence is a reality in many different industrial settings. In practice and in its most general form, optimization is a broad area that encompasses entire fields and many subareas. Today, the term "optimization" seems to be most commonly used when there exists a more or less clear (but at a first glance often hopelessly complicated) mathematical formulation of the problem to be optimized. Nonetheless, the term applies just as well to less rigorous optimization approaches.

A *combinatorial optimization problem* can be loosely defined as an optimization problem in which the set of feasible solutions is discrete [173, 186]. This thesis is concerned with obtaining practical solutions for three industrial combinatorial optimization problems in the areas of embedded real-time systems and condition-based maintenance.

1.1 Real-world Optimization

When applying optimization methods to real problems, several practical issues emerge. First and foremost, it is significant that many industrial-size optimization problems (and indeed two out of three problems in this thesis) do not seem solvable, due to their complexity and size, to the absolute optimum — at least not without a substantial effort to find and "tune" the right method. In practice,

however, other aspects, such as optimization response time, model correctness and the possibility to work interactively with the optimization tool (not to even mention budget limits imposed on the development project), can be equally or more important than finding the absolute optimum.

In addition, there are several other important issues that have not always been treated with the same emphasis as more theoretical problems and solutions. First of all, uncertainties and a lack of accurate information during the development phase often lead to a less than perfect problem model. It might even be that the problem to be solved is not fully understood. Related to this issue is the question of realism of the chosen optimization model. Since the optimization model is by necessity a simplification of reality, the engineers and/or planning personnel using the system frequently have knowledge of circumstances that are not even present in the optimization model. Many users react with disapointment when realizing that the optimization model is a simplification of what is considered the real problem, and therefore does not produce the best possible solution. The consequences of the two issues include inadequate tool support and a less efficient planning process.

The issues above arise too often in practice to be ignored. In the best case, the effects can be that planners compensate by starting follow an experimental optimization approach based on trial-and-error until an acceptable solution is found. In the worst case, the optimization approach may, after much time and effort has been spend in developing and deploying it, prove unusable in practice. Since all models are in practice simplifications of reality, it can appear that there is little that can be done. However, by making sure that the chosen optimization model and associated working process captures most of the relevant side constraints, the risk of deployment failure can at least be reduced. The technical or practical solution for ensuring that the model is accurate enough is less important. However, a close and continuous collaboration in terms of discussing proposed models and solution approaches is recommended. In many cases, conceptually simple solutions such as multi-phase optimization, pre- or post-processing or even manual actions can be preferable for handling side constraints. Planning software users should also be allowed to "tweak" a resulting solution using some form of sensitivity analysis, so that additional knowledge of a situation can be taken into account without modifying the software. In addition, the application should ideally be designed so that additional constraints and features can be added with as little effort as possible.

Naturally, even if the issues above are not applicable, for many real-world problems there currently exist no complete solution methods guaranteeing optimality within reasonable time limits. The goal of this thesis has therefore

been to develop optimization methods *useful in practice* for three industrial combinatorial problems; run-time stack analysis, response-time analysis and condition-based maintenance scheduling and planning. The core requirements for this objective to be met were the following.

Practicality, in that the proposed solutions should be applicable for real problems while including the application-relevant side constraints.

Scalability, in that it should be possible to obtain acceptable solutions for problems of realistic size for the intended application.

Responsiveness, in that the optimization or analysis should not take too long to run, so that users can experiment with the optimization software. However, requirements on optimization response time obviously differ between different application domains and end users. The main goal has been that the optimization software should be able to produce solutions within at most a few minutes, which we judged to be the upper time limit for the studied applications.

Cost effectiveness, in that the optimization methods should be scalable, but not take too much time and effort to develop.

The ambition of the work presented in this thesis was that it should have substantial practical impact. We claim that this ambition has been fulfilled, considering that it has resulted in two different deployed applications within industry. Methods for run-time stack dimensioning presented in Papers A and B have been implemented and integrated into the commercial Rubus development environment by Arcticus Systems AB[1] (see [117]), and a software application for maintenance scheduling based on Papers E and F has been delivered and is currently in use at Siemens Industrial Turbomachinery AB. Although both of these have yet to see wider use, it is, at least to us, clear that the practical utility of the methods and resulting tools have been demonstrated. On the other hand, Papers C and D present methods that have not yet been applied directly in industry. However, the methods in both papers have practical value, and discussions regarding industrial deployment are ongoing. Furthermore, some of the ideas regarding maintenance scheduling found in Paper D were later developed into what is presented in Papers E and F, which motivates its inclusion in the thesis.

[1] Web page: http://www.arcticus-systems.com

1.2 Problem Overview

The thesis is concerned with solving practical problems within industry and academia. In particular, the objective of the thesis work was to propose solutions that were useful in practice and could yield a clear, well-defined benefit. In the different projects this thesis springs from, there has been a clearly identified problem area and customer. This has placed constraints on the choice of methodology and chosen techniques. The following three main problems are addressed in the thesis:

1.2.1 Real-time System Stack Analysis

Many embedded computer systems are safety-critical, in that erroneous behavior can cause physical damage and possibly loss of life. At the same time, producers strive to increase margins by reducing production costs (and actual product costs), such as the cost of hardware, as much as possible. The over-dimensioning of hardware such as CPU:s and RAM is not only costly, but does not in many cases add any value whatsoever to the intermediate and/or end users. For example, an embedded real-time control system equipped with 1M of memory does not add any value (in terms of functionality or performance) over the same system equipped with 16k memory, except for the additional cost of the hardware.

However, ensuring that hardware is not *under-dimensioned* is an issue of high significance. If the hardware used is under-dimensioned, temporal and functional correctness of the system can be compromised. This can be highly dangerous in safety-critical areas such as automotive and avionic applications. One example where under-dimensioning can lead to malfunctioning is the *execution stack*. The execution stack in a software application is used to store local variables, parameter values and return addresses, and can grow and shrink depending on application-specific behavior. If the allocated stack is not large enough, the program will read and write unallocated memory space, which typically leads to an application crash or unspecified application behavior. This can have devastating effects in a safety-critical environment, which is why guarantees regarding stack allocation are important.

In traditional real-time systems, each thread of execution has an individual execution stack. In systems with a large number of threads, a large number of stacks are consequently required. Hence, the total amount of RAM needed for the stacks can grow quite large. A common feature of many real-time operating systems is that they employ *stack sharing*, in which a global run-time stack is shared among the tasks in the system, thereby reducing the amount of RAM

needed. In Papers A and B, we address the problem of how to obtain safe and tight upper bounds on shared stack space in systems with one or more shared execution stacks. The proposed solutions make it possible to reduce costs by using only the amount of RAM actually needed for correct application behavior. At the same time, our algorithms guarantee that a correctly specified real-time system will never run out of stack space as long as the amount of memory that the analysis recommends is made available for the execution stack.

1.2.2 Best-Effort Response-Time Analysis

Response-time analysis is the process of obtaining an estimate of the time from an event to when the processing of that event is finished. The worst-case response time (WCRT) of a task is the highest possible response time for any instance of that task. Using traditional response-time analysis (RTA), it is (under certain assumptions) possible to obtain safe *upper* bounds on the WCRT of one or several tasks in a system. However, many embedded systems break the assumptions of basic RTA by containing code features such as unbounded loops and task interaction affecting response time. In addition, data-dependent execution times may lead to pessimistic response-time results. For these systems, it is difficult or even impossible to obtain a tight and safe upper bound on task response time using RTA.

As a complement to RTA, *best-effort worst-case response-time analysis* can be employed to find a *lower* bound on the actual worst-case response time. Best-effort worst-case response-time analysis involves measuring the real or simulated specific response times of the system given a large set of sample inputs, in order to provoke the system to show its worst behavior (with regard to response time). In most cases, both the best-effort worst-case response-time and the upper bound worst-case response time are inexact, and the true worst-case response time usually lies somewhere in between.

Measurements of response time are often performed using an instrumented executable for the target system [157]. However, for large systems with complex behavior and long response times, running the actual system may be time-consuming. Considering that the number of samples often can be in the order of thousands or even millions, the total evaluation time can easily become prohibitively long. Therefore, it has previously been proposed [35, 118, 139–141] instead to analyze simpler but still detailed models of the target system. Using simulation, the evaluation time of a single sample can be improved significantly, therefore allowing more samples to be examined. In Paper C, we address the problem of provoking the system to exhibit long response times

by controlling the system input parameters using a local search method. The method is based on hill-climbing with random restarts using a representation suitable for capturing the essential properties of the problem.

1.2.3 Dynamic Maintenance Scheduling

In an industrial setting, breakdowns can have a significant impact on short and long-term profitability. During a breakdown, fixed costs of equipment, real estate, labor, etc. remain constant while production is essentially zero. Preventive maintenance, which aims at avoiding breakdowns by periodic inspections and servicing in an effort to capture developing faults early, is therefore an important activity.

However, maintenance is an often underdeveloped area, which in turn often implies excessive maintenance costs. For example, Wireman [278] claims that up to 1/3 of maintenance costs are unnecessary. Estimates of maintenance expenses range from 15–70 % of the total production cost [33, 53, 155]. It is also common that maintenance schedules are constructed mainly for the warranty period of the product. Newly developed products almost always experience a burn-in period with an increased failure rate. This supports a conservative approach when constructing maintenance schedules. The warranty period is also special in that during this period, the manufacturer has liabilities with regard to product functionality. When the warranty period ends, the maintenance schedules are often reused without revision. The sub-optimality of this practice becomes clear when considering that the lifetime of industrial machinery can reach 30 to 40 years. A careful analysis after a run-in period can therefore often reduce maintenance costs significantly with only marginal effects on reliability.

In addition, maintenance is rarely scheduled and planned in conjunction with production. Since maintenance usually has a negative effect on production, it should ideally be coordinated and planned so that the effects on production are minimized, while the maintenance costs are kept below an acceptable level. Today, maintenance is often scheduled once, and the short and medium-term maintenance optimization is left to the person in charge of short-term maintenance planning. Although maintenance planning personnel in many cases perform a remarkable job in making sure that maintenance is done according to plan while taking care of daily disturbances, it is unrealistic to expect planning personnel to produce optimized plans with regard to a life-cycle cost perspective. The information load is also expected to increase with the inclusion of condition monitoring and condition-based maintenance. There-

fore, we are convinced that decision support tools are important to control the maintenance planning process.

In Papers D–F, we study the problem of how to carry out preventive maintenance as efficiently as possible, evaluating potential short-term profits in an overriding life cycle cost perspective. The goal of preventive maintenance optimization is to minimize total costs while still maintaining assets according to maintenance requirements. Reliability data regarding component lifetimes is in general of limited quality or even non-existent, especially for new components. On top of that, there are many other business factors influencing maintenance interval length for commercial equipment. Therefore, we have opted for a deterministic maintenance model where we assume that the risk of failure is negligible for preventive maintenance done within the interval. The proposed solution performs maintenance schedule optimization using heuristic methods, and has been estimated to save substantial costs in practice.

1.3 Thesis Outline

The thesis is organized as follows.

Part I contains an introduction to the thesis and background material.

Chapter 1 gives an informal introduction to the thesis, together with a problem description, the goal objectives, and the thesis contributions.

Chapter 2 contains an introduction to the theoretical topics related to this thesis, including graph theory and algorithms, combinatorial optimization and search, complexity, and mixed-integer linear programming.

Chapter 3 introduces real-time systems, scheduling, response-time analysis, handling of shared resources, and related work on stack analysis and best-effort response-time analysis. The chapter serves as an introduction to Papers A, B and C.

Chapter 4 discusses maintenance practices for gas turbines and rail vehicles, including maintenance policies, availability and reliability, condition-based and reliability-centered maintenance, and maintenance optimization. The chapter serves as an introduction to Papers D, E and F.

Chapter 5 outlines the academic and industrial contributions of Papers A–F, and lists the author's publications included in this thesis, as

well as publications related to his doctoral studies but not included in the thesis. The chapter also concludes the thesis by discussing future work.

Part II contains the six papers on the topics of run-time stack analysis, best-effort response-time analysis and condition-based maintenance planning, which constitute the main academic contribution of this thesis.

Paper A presents a new method to compute the amount of stack memory used in a real-time system. We consider preemptive systems in which some of the tasks can share a single run-time stack, and provide an exact problem formulation, based on run-time properties, which is applicable for any preemptive system model. The main contribution is that we show how it is possible at compile-time to safely approximate the exact stack usage for a commercially available system model: A hybrid, statically and dynamically, scheduled system. Comprehensive evaluations show that our technique can significantly reduce the amount of stack memory needed. A decrease in the order of 70% is typical in the evaluation.

Paper B extends Paper A by considering a more generic task model and by presenting two new methods to bound the stack memory. The first method is a branch-and-bound search for possible preemption patterns, and the second approximates the first in polynomial time. In addition, precedence relations are considered. We evaluate the new methods and previous approaches on random task sets and compare them with each other. The evaluation shows that our exact method can significantly reduce the amount of stack memory needed in the more generic system model considered.

Paper C presents an efficient best-effort approach for response-time analysis, based on the well-known hill-climbing metaheuristic. We target complex industrial systems where response-time measurements or simulation is the only option. A simple yet novel hill-climbing algorithm, where controlled randomization is added in the form of full and partial random restarts, is used to generate input data to a simulated real-time system, where priorities, preemptions and task communication are taken into account. In a thorough evaluation on three models constructed from existing industrial systems, the new algorithm is compared to the current state-of-practice (Monte Carlo simulation) and a previously proposed method. The pro-

posed method was found to be more accurate and on average 42 times faster than the second-best method.

Paper D propose to use online maintenance planning in order to avoid the frequent train service interventions which is often associated with condition monitoring. A dynamic planning software application is used to quickly find new train circulations adapted for the current maintenance requirements of a fleet of vehicles, and the number of maintenance stops is minimized using a heuristic for dynamic packaging of maintenance activities. At the same time, we actively keep the risk of breakdowns low. An evaluation using real-world timetables and vehicle plans shows that the number of service interventions can be reduced significantly compared to traditional cyclic maintenance.

Paper E builds on Paper D and describes and evaluates a novel condition-based gas turbine maintenance strategy. The basis of the strategy is that maintenance should be repeatedly re-optimized to fit into the time intervals where production losses are least costly and result in the lowest possible impact. A prerequisite is that accurate dynamic lifetime estimates are available. The approach is evaluated on a gas turbine used in a real-world scenario, where input from operation data, maintenance schedules and operator requirements are taken into account. In the evaluation, typical cost reductions range from 25 to 65 %, and the calculated availability increase in practice is estimated to range from 0.5 to 1.0 %.

Paper F builds on Paper E and describes the implementation and deployment of the optimization tool. The optimization problem is formally defined, and we argue that feasibility in it is NP-complete. We outline a heuristic algorithm that can quickly solve the problem for practical purposes. We also compare the algorithm with mixed-integer linear programming, and discuss the deployment process of the application. Compared to a mixed integer programming approach, our algorithm is not optimal, but is much faster.

Chapter 2

Combinatorial Problems

This chapter contains a review of graph theory and combinatorial optimization methods and techniques used in this thesis. For much more in-depth information, the interested reader is referred to books by West [272], Golumbic [104] and McKee and McMorris [165] for basic and intermediate graph theory. The book by Papadimitriou [187] contains an extensive overview of computational complexity, while Garey and Johnson [95] provide an accessible overview of **NP**-completeness and related topics. Constraint programming is covered in several books, including [18, 158, 257], while most well-known AI-based search methods are described in [214]. For integer and linear programming, the reader is referred to books by Nemhauser and Wolsey [173] and Papadimitriou and Steiglitz [186]. An introduction to mathematical programming with many examples is also given by Winston [277].

2.1 Graph Theory

A *graph* is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of *edges* connecting two vertices. In a *directed* graph, the set of edges are directed, i.e., on the form (v_i, v_j), and referred to as *arcs*. In an *undirected* graph, each edge can be represented by two arcs (v_i, v_j) and (v_j, v_i). Vertices connected by an edge are said to be *adjacent* or *neighbors*. A *complete* (undirected) graph is fully connected in that all vertices are directly connected to all other vertices.

A graph H is a *subgraph* of another graph G if H's vertex set is a subset of G's vertex set and if H's edge set is a subset of G's edge set, restricted to the vertices in H. A subgraph H of G is *induced* if it contains all edges in G connecting nodes in H. A *path* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k \in V$

such that there exist arcs $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k) \in E$. A path is a *cycle* if in addition there exists an arc $(v_k, v_1) \in E$. A graph is *acyclic* if it does not contain any cycle.

A *weighted* graph is a graph with an additional weight function $w(e)$ mapping edges to weights. The *distance* of a path $v_1, v_2, \ldots, v_k \in V$ in a weighted graph is $\sum_{i=1}^{k-1} w((v_i, v_{i+1}))$. A *shortest path* between two nodes is a path for which no other paths exist between the two nodes with a lower distance. For graphs with non-negative weights, a shortest path can be found in $O(D \cdot |E| + X \cdot |V|)$ using Dijkstra's algorithm [57, 76]. Here, D and X represent the time needed to maintain a queue of vertices sorted according to shortest distance to the source node; D is the time needed to decrease the distance of a vertex, and X is the time to extract and remove the vertex with lowest distance from the queue. If the queue is implemented using an efficient data structure such as a Fibonacci heap [89], the amortized time complexity of Dijkstra's algorithm becomes $O(|E| + |V| \log |V|)$.

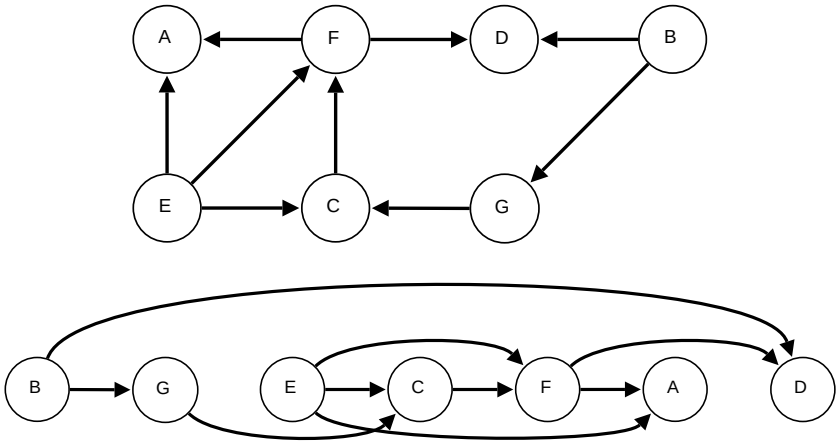


Figure 2.1: A directed acyclic graph (top) and one of its topological orderings (bottom).

In Papers A and B, we are interested in finding the *longest* paths within certain induced subgraphs of a graph representing the possible preemptions that can occur within a set of tasks. Since shortest paths can easily be found in graphs with non-negative weights, it may come as a surprise that finding longest paths is much harder. In fact, the longest path problem is **NP**-complete

for generic graphs [258]. Fortunately, it turns out that for directed acyclic graphs (DAGs), both shortest and longest paths can be found in $O(|E| + |V|)$ time. The method is based on first performing a *topological sort* [127, 247] and then processing the vertices in topological order (as described in [57]). A topological ordering of a directed acyclic graph is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges; an example is shown in Figure 2.1. Note that every DAG has one or more topological orderings.

Algorithm 2.1 produces a topological ordering of a DAG based on depth-first search (DFS). The algorithm is from the book by Cormen *et al.* [57], but is originally due to Tarjan [246]. An alternative was described by Kahn as early as 1962 [127]. Algorithm 2.1 has its entry point in the function DFS and loops through each node of the graph in an arbitrary order, initiating a DFS that terminates when it hits any node that has already been visited since the beginning of the topological ordering. The array $visited$ is used to keep track of which vertices has been visited so far, and L is the list of vertices in inverse order; both are updated during the execution of the algorithm. Finally, u is the current vertex and v is a vertex adjacent to u.

Algorithm 2.1: Topological sort using depth-first search.

```
VISIT(u, visited, V, E, L)
(1) if visited[u] = false
(2)    visited[u] ← true
(3)    foreach (u, v) ∈ E
(4)       VISIT(v, visited, V, E, L)
(5)    L ← L ∪ {u}

DFS(V, E)
(1) L ← ∅
(2) foreach u ∈ V
(3)    visited[u] ← false
(4) foreach u ∈ V
(5)    VISIT(u, visited, V, E, L)
(6) return L
```

Given a topological ordering, a linear time algorithm for finding longest (and shortest) paths can then be obtained by processing vertices in the topological order, updating distance labels of adjacent nodes accordingly. The method we use in Papers A and B is given in Algorithm 2.2, and takes a list L of

the vertices in the graph in topological order, the set of edges E and a weight function w, and returns the longest path distance $l(u)$ from any node to any node u, and the predecessor node $p(u)$ in one such path, where ε is used to denote the absence of a predecessor. Note that the linear time complexity of the algorithm is important in applications where longest paths must be found repeatedly. For example, in Papers A and B, $|V|$ longest paths must be found in different induced subgraphs, yielding a total time complexity of $O(|V| \cdot |E|)$.

Algorithm 2.2: Longest paths algorithm for a topologically sorted DAG.

LONGESTPATHS(L, E, w, l, p)
(1) **foreach** $u \in L$
(2) $\quad l[u] \leftarrow 0$
(3) $\quad p[u] \leftarrow \varepsilon$
(4) **foreach** $u \in L$
(5) $\quad$ **foreach** $(u, v) \in E$
(6) $\quad\quad$ **if** $l[u] + w(u, v) > l[v]$
(7) $\quad\quad\quad l[v] \leftarrow l[u] + w[u, v]$
(8) $\quad\quad\quad p[v] \leftarrow u$

2.1.1 Cliques

In the previous section, we briefly discussed the technical solution in our approach to stack analysis — presented in Papers A and B — which consists of repeatedly searching for longest paths within certain induced subgraphs of a preemption graph (defined in Paper A, and illustrated in Figures 5.1 and 5.2). The induced subgraphs in which we will be searching for longest paths later on are called *cliques*. They consist of subgraphs where each vertex is connected to every other vertex. Formally, a clique is then a complete subgraph. A clique is *maximal* if it cannot be extended with any node. The problem of finding a maximal clique is in general **NP**-complete [272]. However, polynomial-time algorithms are known for certain types of graphs. In this section, we will review two graph families where this property holds, and where the number of maximal cliques are also bounded linearly.

The following assumes that we have an undirected graph. A *chord* is an edge between non-consecutive vertices in a cycle. A cycle, together with the edges in the cycle, which are also an induced subgraph, is then *chordless*; since it is an induced subgraph, it only contains the edges in the cycle, and can therefore not contain a chord. Note that all cycles of length 3 are chordless. A graph is *chordal* if and only if it contains no chordless cycles. A chordal graph is

illustrated in Figure 2.2. Chordal graphs are sometimes referred to as *triangulated* graphs, and chordless cycles of length 4 or more are simply called *holes*. A *perfect elimination order (PEO)* is an ordering of the vertices V in a graph

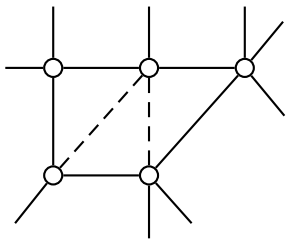


Figure 2.2: A cycle (solid) with two chords (dashed). The subgraph shown is chordal, but removing any of the dashed edges would result in a chordless cycle of length 4.

such that each vertex v forms a clique together with all adjacent vertices occurring later in the ordering. A graph has a PEO if and only if it is chordal [90]. Chordal graphs can be recognized in linear time by finding a PEO, which can be done using lexicographical breadth-first search (BFS) [58,212] in $O(|V|+|E|)$. Given a PEO, the set of maximal cliques can be found by testing each PEO-induced clique for maximality [165]. In addition, chordal graphs can have at most $|V|$ maximal cliques [272]. Both these properties are useful when obtaining a safe upper bound on stack usage later in Papers A and B. In an *interval graph*, the vertices can be represented by intervals $[a, b]$ in a single dimension, and the edges correspond to interval intersection. In other words, there is an edge between two intervals $[a, b]$ and $[c, d]$ if and only if $a < d$ and $c < b$. All interval graphs are chordal; a PEO is given by ordering the vertices after their interval "start point" (i.e. according to a in the interval $[a, b]$), breaking ties arbitrarily.

A simple algorithm (shown in Algorithm 2.3) for finding maximal cliques in an interval graph is based on processing two queues L_s and L_e consisting of the intervals sorted by start and end time, respectively. The algorithm is in either *build* or *break* mode, starting in break mode. For each start point, the corresponding vertex is marked as active, and the mode is set to "build". If the

current mode is "build" when an end point is scanned, then all active vertices are output as a maximal clique, and the mode is set to "break". A vertex is always marked as inactive when its end point is scanned.

Algorithm 2.3: Maximal cliques in an interval graph.

MAXCLIQUES(V)
(1) $L_s \leftarrow V$ ordered by start time
(2) $L_e \leftarrow V$ ordered by end time
(3) $m \leftarrow$ break, $Q \leftarrow \emptyset, q \leftarrow \emptyset$
(4) **while** $L_s \neq \emptyset \vee L_e \neq \emptyset$
(5) $(a, b) \leftarrow$ FIRST(L_s)
(6) $(c, d) \leftarrow$ FIRST(L_e)
(7) **if** $a < d$
(8) $L_s \leftarrow L_s \setminus \{(a, b)\}$
(9) $q \leftarrow q \cup \{(a, b)\}$
(10) $m \leftarrow$ build
(11) **else**
(12) $L_e \leftarrow L_e \setminus \{(c, d)\}$
(13) **if** $m =$ build **then** $Q \leftarrow Q \cup \{q\}$
(14) $m \leftarrow$ break
(15) $q \leftarrow q \setminus \{(c, d)\}$
(16) **return** Q

2.2 Satisfiability and Optimization

The classical definition of optimization is the process of finding the highest- or lowest-ranked solution to a problem, as measured by one or more *objective functions*. In *multi-objective optimization*, several objective functions exist that should simultaneously be optimized. In this case, several different optimality criteria exist, the most common ones being based on aggregate objective functions and Pareto optimality [79].

In this thesis, we use the term "optimization" loosely in that we also use it for approaches where the absolute optimum is not required. This thesis is also only concerned with single-objective optimization; although aggregate objective functions are used, a natural interpretation of aggregation exists in that the objective is to minimize cost. Obviously, not all optimization problems are of minimization type. However, for maximization problem, if the function to maximize is well-defined, then minimizing the negation of this will maximize the original function.

The sought-after solution to the problem is specified as a set of *variables* that should be assigned values. In *unconstrained* optimization, the problem variables are allowed to take any possible value. In *constrained* optimization, on the other hand, there exist a set of *constraints* that restrict the set of feasible solutions to the problem. For single-objective constrained minimization, the goal is therefore to find a solution that is

1. *feasible*, i.e., satisfies a set of *constraints*, and
2. *near-optimal* in that it, as far as possible, minimizes the cost function.

Constrained optimization can further be divided into several subclasses depending on the type of constraints and cost function used. Some of the most well-known subclasses are

- *linear programming*, in which the cost function is linear, and all constraints are linear inequalities;
- *mixed integer programming*, in which some of the variables have integrality constraints;
- *0–1 integer optimization*, in which all variables are restricted to values of either 0 or 1;
- *quadratic programming*, in which the cost function is quadratic, and all constraints are linear inequalities; and
- *nonlinear programming*, in which the cost function and all constraints can contain nonlinear parts.

A *satisfiability* problem is concerned only with finding a feasible solution for a set of constraints, and can also be seen as an optimization problem with a constant objective. A *combinatorial problem* is an optimization problem where the set of feasible solutions is finite [186]. Combinatorial problems are abundant in all areas where discrete resource-constrained problems either appear naturally or when a discretization of an otherwise continuous problem may be beneficial. Examples of well-known combinatorial problems include the vehicle routing problem [52, 60] (an overview can be found in [255]), the traveling salesman problem [17], the knapsack problem [132, 160], the cutting stock problem [65, 101, 190] and the generalized assignment problem [47,213,225]. Some well-known combinatorial puzzles and games include Chess [269] and the related eight queens problem [238], Sudoku [150,282] and Go [36,146,211].

2.2.1 Branch and Bound

When no suitable polynomial-time algorithm exists for a discrete satisfaction or optimization problem, one strategy might be to simply search the entire set of feasible solutions (or a subset thereof) for a solution. This is often referred to as *combinatorial search*. Because combinatorial search can in many cases take exponential time to solve the problem, methods to cut down on the size of the search space have been devised. One of the most common approaches is the *branch and bound method*, in which a tree of partial solutions are explored. The branch and bound method is used in Paper B for performing stack analysis.

The branch and bound method works as follows. For a minimization problem, a *lower bound* on the objective function is established (preferably in polynomial time) for each node in the tree. A lower bound is a function that is always less or equal to the optimal objective value. If the lower bound is greater than the objective function of the best found solution (which in turn is an upper bound on the optimal objective value), then the corresponding node can be removed from the search, since the node lower bound guarantees that no search from that node can ever yield an objective lower than the lowest found so far. If the node is not removed, the search continues by *branching* on that node.

Several branching strategies exist depending on the problem structure and the representation of a solution; common choices include domain splitting (in which the domain of a variable is split into two parts) and variable assignment. Note that branch and bound can be implemented using depth-first search, breadth-first search or other variants; pseudocode for a simple depth-first branch and bound algorithm is shown in Algorithm 2.4. The original branch and bound method was conceived in 1960 by Land and Doig to solve generalizations of linear programming to discrete variables [145].

Algorithm 2.4: Depth-first branch and bound search for non-binary optimization problems with an objective function z.

BB($node$)
(1) **if** LEAF($node$) **then return** $node$
(2) $(s_0, \ldots, s_{b-1}) \leftarrow$ SUCCESSORS($node$)
(3) $best \leftarrow$ NIL
(4) **for** $i = 0$ **to** $b - 1$
(5) **if** LOWERBOUND(s_i) $< z(best)$
(6) $best \leftarrow \text{argmin}_z(best, \text{BB}(s_i))$
(7) **return** $best$

2.2.2 Constraints and Propagation

Informally, a *constraint satisfaction problem (CSP)* can be defined as follows.

> Given a set of constraints, is there an assignment to the variables in the constraints such that all constraints are satisfied?

A special case of a CSP is the *propositional satisfiability problem (SAT)*, which can informally be defined as follows:

> Given a Boolean formula in conjunctive normal form, is there a truth assignment to the variables satisfying the formula?

CSP and SAT have been studied in numerous books and articles, and SAT was the first problem shown to be **NP**-complete [54]. Both CSP and SAT are core problems in computing theory and mathematical logic. Much of the effort spent on constraint satisfaction and satisfiability research can be attributed to the generality of CSP and SAT. In practice, solution methods for CSP and SAT are useful for solving problems in automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, databases, robotics, integrated circuit design, and computer network design.

Conventional constraint satisfaction methods have been shown to work well on a large number of problems from real life, like scheduling, planning and resource allocation problems. Unfortunately, these methods are in general time- and space consuming. Because of this, they are not always suitable. For example, a *dynamic* planning problem is a planning problem where the parameters change during the execution of the plan. This calls for a planner that is able to recover from changes in the plan within reasonable time limits.

Constraint Propagation

Constraint propagation is a technique for search space reduction taken from the area of constraint programming [154]. In its purest form, constraint propagation simply involves removing values from the domains of free variables which are inconsistent with the current partial assignment and the constraints in the problem. Constraint propagation can be described as programmed search where the domains of the problem variables are narrowed iteratively, according to the constraints in the problem, as the search progresses. The two most common propagation approaches are *domain propagation*, where the domains are modeled explicitly as a set of the allowed values, and *interval propagation* [151], where only the interval of the domain is stored. In general, domain propagation prunes the search space more efficient than interval propagation.

However, interval propagation is faster and uses less memory space. Interval propagation is used in Papers E–F to reduce the search space during maintenance optimization. For more information on constraint programming and propagation, see the books by Marriott and Stuckey [158], Tsang [257], and Apt [18]. Except for interval propagation, constraint programming is not considered in this thesis. However, it is worth noting that the methodology has previously been used to formulate and solve many practical scheduling and planning problems (see, for example, work by Kreuger *et al.* [142, 143], Fox [88], Sadeh *et al.* [216, 217], Le Pape [188], Smith *et al.* [234–236], Davenport and Tsang [62] and El Sakkout and Wallace [219]). Scheduling problems are often formulated using specialized global constraints, as described in the articles by Beldiceanu [30] and Régin [209, 210].

2.2.3 Mixed Integer Programming

A linear programming problem (LP) consists of a vector of n real-valued decision variables $x = (x_1, \ldots, x_n)$, a linear objective function z and a set of m linear inequalities as shown below in Equation (2.1)

$$
\begin{array}{lccccccccc}
\max z = & c_1x_1 & + & c_2x_2 & + & \cdots & + & c_nx_n & & \\
\text{subject to} & & & & & & & & & \\
& a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & \leq & b_1 \\
& a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & \leq & b_2 \\
& \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
& a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & \leq & b_m \\
& x_1 & & & & & & & \geq & 0 \\
& & & x_2 & & & & & \geq & 0 \\
& & & & & \ddots & & & & \vdots \\
& & & & & & & x_n & \geq & 0
\end{array}
\tag{2.1}
$$

A feasible solution to Equation (2.1) is an assignment of x that satisfies all m inequalities; an optimal solution is a feasible solution which mazimizes z. In vector notation, a linear programming problem can be defined [173] as

$$\max\{cx : Ax \leq b, x \in R^n_+\} \tag{2.2}$$

where c is a $1 \times n$ matrix, $b = (b_1, \ldots, b_m)$ a m-dimensional vector, A an $m \times n$ matrix containing rational numbers, R^n_+ is the set of nonnegative real

p-dimensional vectors and $x = (x_1, \ldots, x_n)$ is the variable vector. The linear programming problem is most commonly solved using the simplex algorithm [277] created by George Danzig in 1947. Linear programming is in **P**; Khachian [133] derived the first polynomial algorithm ($O(n^5)$) in 1979. Since then, more practically useful polynomial algorithms have been found, starting with Karmarkar's algorithm [128].

A mixed integer programming (MIP) problem can be defined as

$$\max\{cx + hy : Ax + Gy \leq b, x \in R_+^n, y \in Z_+^p\} \tag{2.3}$$

where in addition h is a $1 \times p$ matrix, G an $m \times p$ matrix containing rational numbers, Z_+^p is the set of nonnegative integer n-dimensional vectors, and $y = (y_1, \ldots, y_p)$ is the integer variable vector. The obvious difference between MIP and LP problems are that in MIP, some of the variables are only allowed to take integer values. Perhaps surprisingly, this in general makes MIP much harder. Mixed integer programming is in fact **NP**-complete, even when restricted to binary (0–1) variables [130]. Even so, MIP optimization models have been remarkably successful in representing many real-world problems, and MIP solvers such as CPLEX from ILOG [121] and Xpress-Optimizer from Dash Optimization [61] are considered by many to be the standard tool for solving combinatorial problems. In Papers E and F, we use a MIP solver as comparison against our heuristic approach.

2.2.4 Limited Discrepancy Search

Many industrial problems can be solved using tree search methods, especially if guiding heuristics are available. For example, best-first search methods such as A* search [114, 175] have been successful on many problems. However, A* search relies on the availability of a good admissible heuristic, and if no such heuristic is available, A* search will use too much memory on some problems to be practically useful. Depth-first search methods avoids the memory issues with breadth-first search and A*. However, it can easily get stuck in unproductive areas of the search tree when the heuristic fails. Limited discrepancy search (LDS, [91, 115, 129, 138, 266]) addresses this problem. The basic idea of LDS is to use depth-first search guided by a heuristic, but allow a specified number of so-called *discrepant* choices that disagree with the heuristic. The maximum number of discrepant choices allowed in each path from root to leaf is the *discrepancy* parameter k. The basic LDS procedure, introduced by Harvey and Ginsberg in [115] and further improved by Korf in [138], works primarily on binary search trees, although Harvey and Ginsberg discuss extensions

to non-binary problems. In [266], Walsh improves LDS by first considering discrepancies that occur at the top of the search tree. This is done by introducing a depth limit that is iteratively increased. In [129], LDS is extended to handle arbitrary CSPs, constraint propagation and learning of the variable ordering heuristic. In [91], Furcy and Koenig extend beam search [215, 283] with LDS-type backtracking.

Algorithm 2.5, which is used in Papers E–F, shows an extension of LDS along the lines of the ideas proposed in [129]. The algorithm is also modified to continue searching for a best possible solution as measured by an objective function z, which is infinitely-valued for the empty node NIL. In [115], it is discussed whether all discrepant choices emanating from a specific node should be treated equally, i.e., counted as depleting one unit of discrepancy, or whether each further step away from the heuristic should be counted as using up one more discrepant choices [129]. In Algorithm 2.5, the latter view is taken. Figure 2.3 shows the unique paths explored for each choice of k in a tree with branching factor 3.

Algorithm 2.5: Limited Discrepancy Search for non-binary optimization problems.

LDS-PROBE($node, k, d, b$)
(1) **if** LEAF($node$) **then return** $node$
(2) $(s_0, \ldots, s_{b-1}) \leftarrow$ SUCCESSORS($node$)
(3) $best \leftarrow$ NIL
(4) **for** $i = \max(0, k - (b-1)(d-1))$ **to** $\min(b-1, k)$
(5) $\quad best \leftarrow \text{argmin}_z(best, \text{LDS-PROBE}(s_i, k-i))$
(6) **return** $best$

LDS($node, maxdepth$)
(1) **for** $k \leftarrow 0$ **to** maximum depth
(2) $\quad result \leftarrow$ LDS-PROBE($node, k, maxdepth$)
(3) $\quad$ **if** $result \neq$ NIL **then return** $result$
(4) **return** nil

In [138], Korf improved LDS so that it only generates paths with exactly k discrepancies. This is done by keeping track of the remaining depth d, pruning branches for which $d \leq k$. It is worth pointing out that in modifying LDS to non-binary trees, a similar improvement can be done if the maximum branching factor b is known. At most, d discrepant choices can be made in a subtree of depth d, each choice using up at most $b-1$ discrepancies. Given that choices

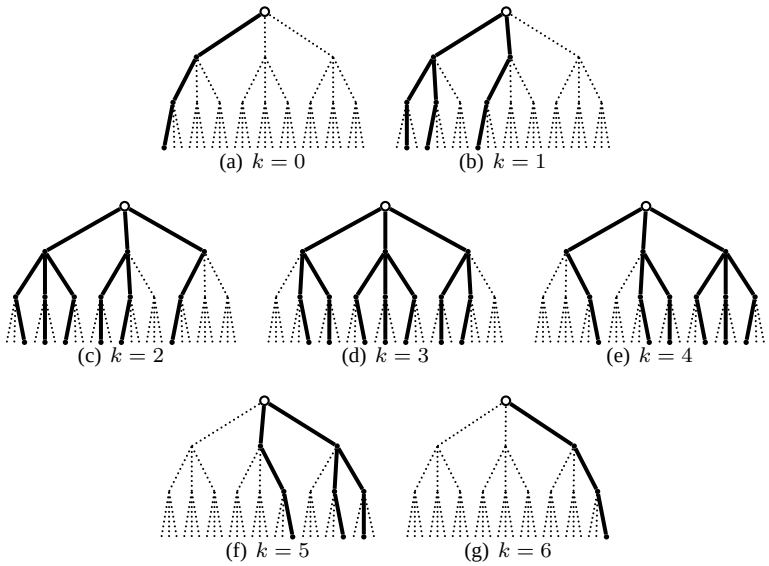


Figure 2.3: Unique explored edges for different discrepancy values in a tree with branching factor 3.

are ordered according to falling heuristic value (so that the preferred choices occur early), we can therefore prune the i^{th} choice if $(b-1)(d-1)+i < k$, or in other words, start with choice number $\max(0, k-(b-1)(d-1))$. The function call SUCCESSORS($node$) returns a list of feasible successor nodes in increasing order of heuristic value. The discrepancy parameter is k.

2.2.5 Local Search

Local search, or *iterative improvement*, is an alternative approach to optimization using complete methods. Local search methods have the advantage that as soon as a feasible solution has been found, this solution is always available during the search. This property is commonly known as *anytime behavior*, and is of particular interest when feasible solutions are easily found, or when the iterative improvement can work solely by manipulating feasible solutions. Another advantage of local search methods is that they can solve certain problems much more efficiently that regular constraint solvers. At the same time, techniques based on local search often have quite modest resource consumption.

In Paper C, we use local search in the form of a hill-climbing algorithm with random restarts to solve the best-effort response-time analysis problem, and in Paper D, a local search procedure is used to find alternative routes for a train in need of maintenance.

In iterative improvement, a given candidate solution is improved in several repeated steps by changing small parts of the solution. The resulting set of candidate solutions is called the neighborhood. The algorithm then proceeds by selecting as the next solution either the first found improving neighbor or one of the best neighbors. In the hill-climbing algorithm [214], the search terminates when a local minimum has been reached. Other methods, such as Tabu search [102] or Simulated Annealing [136] have mechanisms to escape local minima. The initial starting point of the local search is usually generated randomly or using a constructive heuristic.

The neighborhood is one of the parameters that affect the performance of local search algorithms the most. The neighborhood should be chosen so that neighbors who are likely to improve the objective are included. However, if the neighborhood is too large, the local search procedure will consequently spend a large amount of time exploring it, especially if the neighbor that improves the objective value the most is wanted. If it is also non-trivial to compute the objective value of a candidate, the exploration process in itself can take a significant amount of time. One example is the combined problem of maintenance routing and scheduling presented in Paper D, where the cost of a train circulation depends on the maintenance schedules of the train units. Since the maintenance schedules are also dependent on the circulations, new maintenance schedules have to be found to evaluate neighbors. A common technique for evaluating neighbors more efficiently is therefore to compute only the cost difference that the transition yields. This is sometimes called incremental computation [10]. In the planning algorithm outlined in Paper D, we update costs by observing that only the maintenance schedules of the trains where the neighboring circulation differs needs to be recomputed.

It is generally agreed that randomization may help local search procedures overcome local minima. Stochastic behavior may be introduced in numerous ways, one of the most basic being to introduce *random restarts* in the search after a fixed number of transitions. Walser [265], Selman et al. [229], Gent and Walsh [97, 98] and Gu et al. [107] take this approach, as do we in Paper C. Another common randomization strategy is to introduce *random walk* in the search. Random walk is the occasional random transition (or transitions) in the search space, the probability taking a random transition depending on a parameter typically supplied by the user of the search algorithm. A third possibility

is to change the neighborhood of an assignment according to a probabilistic distribution. In the well-known WalkSAT algorithm, introduced by Selman et al. [229], a successor assignment is selected by picking an unsatisfied clause at random, and from this clause selecting promising variables. In Paper C, we introduce randomness in the neighbor selection by only considering a randomly-chosen subset of the full neighborhood obtained by changing a single variable value.

2.3 Computational Complexity

Computational complexity [187] is an area of computer science and mathematics concerned with time and space consumption, the expressive power of different computational mechanisms and the related complexity classes. In this section, we will give a short summary of the theory of **NP**-completeness. The interested reader is referred to [95, 187] for more information. We will use the theory in Paper F when proving **NP**-completeness of the maintenance scheduling problem.

2.3.1 Decision Problems

A *decision problem* is a problem whose answer is "yes" or "no". A typical example is SAT, introduced in Section 2.2. Other examples are:

Knapsack Given a knapsack of unit capacity and a set of n items, each item i having a size $a_i \in (0, 1]$ and a *value* b_i, is there a subset of items fitting in the knapsack with a higher value than a given lower threshold k?

Bin-packing Given a set of k bins of unit capacity and a set of n items, each item i having a size $a_i \in (0, 1]$, is there a packing (i.e., assignment of items to bins) of all items using the available bins?

2.3.2 Complexity Classes

It is common to adopt a view in which decision problems are *languages* (i.e., subsets of binary strings $0, 1^*$ in which the language consists of all strings that encode a "yes" problem instance). A language $L \in \mathbf{P}$ if there exists a deterministic polynomial time bounded Turing machine M that can decide L, i.e., for each string $x \in \{0, 1\}^*$:

- if $x \in L$ then $M(x)$ accepts, and
- if $x \notin L$ then $M(x)$ rejects.

Intuitively, the class **P** corresponds to problems that can always be efficiently solved. A language $L \in$ **NP** if there is a polynomial p and a polynomial time bounded Turing machine M such that for each string $x \in \{0,1\}^*$:

- if $x \in L$ then there is a string y of polynomially bounded length, i.e., $|y| \leq p(|x|)$, such that $M(x,y)$ accepts, and
- if $x \notin L$ then for any string y such that $|y| \leq p(|x|)$, $M(x,y)$ rejects.

The string y that helps verifying that x is indeed a "yes" instance is called a *solution* to the problem; thus, **NP** is the class of problems that have short and quickly verifiable solutions. As an example, given a knapsack instance and a proposed solution in the form of a subset of items, it is easy to check whether this subset 1) fits inside the container, and 2) has a value higher than k. Clearly, the knapsack problem is in **NP**.

A language L belongs to the class **co-NP** if and only if $\overline{L} \in \mathbf{NP}$; thus, **co-NP** is the set of problems that have short, quickly-verifiable "counterexamples". For instance, the language L of prime numbers allows for counterexamples in the form of factorizations for a number n, which is proof that $n \notin L$.

2.3.3 Reductions

Let L_1 and L_2 be two languages in **NP**. Then, L_1 *reduces to* L_2 if there is a polynomial time deterministic Turing machine T that, given a string $x \in \{0,1\}^*$, outputs a string y such that $x \in L_1$ if and only if $y \in L_2$. In other words, T translates problem instances of type L_1 into instances of type L_2. As a consequence, if L_1 reduces to L_2 and L_2 is polynomial time decidable, then so is L_1. This type of reduction is also called a *polynomial-time many-one reduction, polynomial transformation* or *Karp reduction.* Reductions of this type are very useful in proving **NP**-completeness.

2.3.4 NP-Completeness

A language L is **NP**-*hard* if every language $L' \in \mathbf{NP}$ reduces to L. A language is **NP**-*complete* if $L \in \mathbf{NP}$ and L is **NP**-hard. An **NP**-complete language L is a hardest language in **NP** in that a polynomial time algorithm for L implies that there exist polynomial time algorithms for every language in **NP** (i.e., **P**=**NP**).

Once one problem L has proven to be **NP**-hard, other problems can be established as **NP**-hard by giving polynomial-time reductions from L to these problems. SAT was shown in [54] to be **NP**-hard; the proof idea is to show that for any language L in **NP**, there exists a deterministic polynomial-time Turing

machine that can translate strings $x \in \{0,1\}^*$ into SAT formulas f such that the existence of a truth assignment satisfying f implies that $x \in L$.

2.4 Summary

This chapter reviews some topics in graph theory, combinatorial optimization and complexity, which are useful for understanding the methods employed in Papers A–F. In graph theory, algorithms for computing longest paths in directed acyclic graphs and cliques in interval graphs (which are special cases of chordal graphs) are described as background material for Papers A and B. Some different methods for combinatorial problem solving used throughout this thesis were outlined, including local search, which is used in Papers C and D, and Limited Discrepancy Search in particular, used in Papers E and F. The chapter also contains some basic complexity theory, which is used in proving **NP**-completeness of the maintenance scheduling problem presented in the last two papers of the thesis.

Chapter 3

Real-Time Systems

Using computers for process control and physical interaction is becoming more and more common in areas where control has previously been provided entirely by mechanical or electrical means. An *embedded system* is a computer system that is part of a larger system, performing some of the functions of that system [120]. Their role is often to replace a traditional mechanical solution, thereby reducing production costs, increasing efficiency and enhancing functionality of the product.

Embedded systems are almost ubiquitous in nearly all of today's technologically focused industry, including the telecommunication, automation, aircraft, automotive and railroad industries. According to Hansson *et al.*, the software account for a major part of the value growth in the automotive industry [113]. A classical example of an embedded system is the computers controlling critical functionality in a road vehicle, such as lock-free brakes, steering, ignition, airbags and traction control. Embedded systems are also becoming an increasingly important part of our daily lives. For example, nearly all consumer electronics on the market contain one or more computational units, providing extended functionality, intelligent behavior and overall ease of use. One motivation for using an embedded system is often reduction of costs. Another is the addition of advanced functionality, which would not be possible without computers. In a typical modern automobile, embedded systems manage, e.g., driving assistance, information, and entertainment features [223]. This implies that both safety-critical features and less critical features need to be managed in the same system. Figure 3.1 illustrates the complexity level of electronic systems that has been reached in a modern automobile. The Volkswagen Phaeton

shown in the figure contains 11,136 electrical parts, 61 ECUs (of which 31 can be externally diagnosed), an optical bus for infotainment data, sub-networks, and 3 CAN-buses connecting 35 ECUs. Communication-wise, there are approximately 2,500 signals in 250 CAN messages [149].

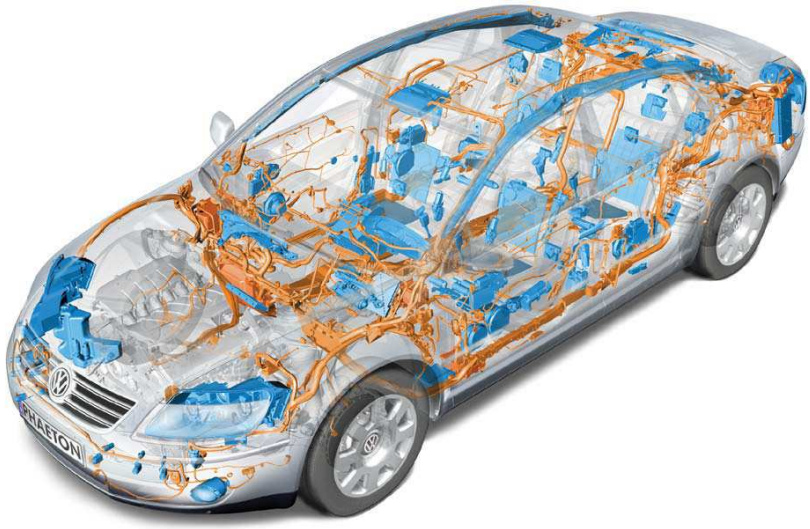


Figure 3.1: The electronic systems (blue) and communications channels (orange) in a Volkswagen Phaeton automobile (Image is from [149] and is courtesy of Volkswagen AG).

A real-time system (RTS) is a system in which *timeliness* is equally important for the system to work properly as the *functional correctness* of the implementation [241]. A common misconception is that an RTS is a system that responds quickly [240]. Instead, an RTS can be defined as a system with a temporally *predictable* response. As an example, the triggering of an automobile airbag is an application in which timeliness is crucial. Obviously, triggering the airbag too late is disastrous for the passenger. However, triggering the airbag too *early* is equally fatal, since the airbag is only inflated fully for a very short period of time. Inflating the airbag too early will therefore result in the airbag being partially deflated at the time the driver or passenger

hits the airbag.

Many real-time and embedded systems are resource-constrained in that the resources available to perform its intended functionality are limited. A typical processor used in an embedded system is much less powerful than the processor used in a desktop computer. Also, the amount of memory available is often measured in kilobytes instead of gigabytes. In addition, since many embedded systems are safety critical, we must be sure that the limited resources available are enough for sufficient system safety and correct temporal and functional behavior. One such limited resource is computational capacity. If the application requires more computational capacity than what is available in the form of the processor being used, the application will most likely exhibit a temporal behavior the system has not been designed for. Going back to the previous example of an automobile airbag, this could result in the airbag deploying too late. Another example of a limited resource is the memory available for the application. If the application requires more memory than is available, data corruption and/or a program crash can result. Both situations can be considered fatal; a consequence of either can be that the program may not function as intended, which is potentially devastating in a safety-critical application. Therefore, analysis methods are needed to guarantee the predictable behavior of the system. To be useful in practice, such methods also have to be suitable for industrial use, in that they should be easy to use and have an acceptable computational complexity.

3.1 Real-Time Operating Systems

A real-time operating system (RTOS) is an operating system which is specifically engineered for real-time applications. In practice, this means that the operating system has functionality for fulfilling application timing constraints. RTOS:s usually provide services such as real-time scheduling, mutual exclusion and intra-system communication. In general, an RTS consists of a set of processors running an RTOS and interacting over one or several communication networks, such as CAN [64,122,176,179,252], FlexRay [87] or TTP [137]. On each processor, a set of *tasks* are executed. The tasks, representing the different computations to be performed by the application, are dispatched by a scheduling algorithm of the underlying RTOS. A specific invocation of a task is called a *task instance*, or just *instance* for short. Each task has an *execution time*, which is in general dependent on the environment it is executing in. This includes factors such as processor speed (varying dynamically with voltage in many newer processors), processor and bus caches, memory latency, communi-

cation delays and state and input parameters of the task. All of these can affect execution time. The exact execution time for all possible situations is therefore either hard or impossible to obtain. A common approximation is to use a *worst-case execution time (WCET)* estimate, assumed to be a safe approximation of the actual execution time under all circumstances. WCET estimates can be found by measurements and/or simulation (see for example work by Edgar and Burns [78], the commercial tool *VirtualTime* [203] and *ARTISST* [67]) and static analysis methods [81–83, 195, 196, 276]. Tools based on abstract interpretation, such as Bound-T [248] and AbsInt [3], can also be used for WCET analysis of software systems.

Timing constraints are abundant in RTS:s. Some of the most commonly used timing constraints include *release time* and *deadline* constraints, specifying the earliest invocation time and the latest finishing time of a task respectively. Tasks invoked repeatedly with a fixed time interval (e.g. the wheel speed of an automobile should be sampled every millisecond) are called *periodic tasks*, while tasks lacking a predetermined inter-arrival time are called *aperiodic tasks*. Finally, tasks that can arrive with a *minimum inter-arrival time*, but also less frequently, are called *sporadic tasks*.

3.1.1 Scheduling

One of the most important parts of an RTOS is the scheduling algorithm it uses to dispatch tasks. The scheduling algorithm determines in what order arriving tasks should execute by dispatching tasks ready for execution. There exist a plethora of scheduling algorithms [46, 230], and this thesis only gives a short overview. Scheduling is divided into *offline* and *online scheduling*. In the offline scheduling approach (e.g., [80, 201, 280]), scheduling decisions are made before system deployment, and result in a fixed schedule. At runtime, the dispatcher simply schedules tasks according to the predetermined schedule. Although most suitable for periodic execution, offline-scheduled systems have the advantage of full predictability; the schedule can be engineered to fulfill almost any imaginable temporal constraint in advance of deployment. For examples of temporal constraints that can be fulfilled by an offline scheduler, see [80]. Once the schedule has been developed, verified and tested, the temporal correctness is guaranteed as long as WCET estimates are safe. On the other hand, offline-scheduled systems can be inflexible with regard to software maintenance and continuous development. In essence, adding a single task to an RTS can result in the full schedule needing to be rebuilt.

In online scheduling (e.g., [152]), tasks are scheduled at runtime depend-

ing on the current state of the system. The schedule is therefore generated continuously during the execution of the RTS. In essence, an online scheduling algorithm dispatches the task that has the highest *priority* of the tasks in the ready queue. Therefore, online scheduling algorithms differ mostly in how tasks are prioritized. In *fixed priority pre-emptive scheduling* (FPS, [20, 21, 44, 131, 148, 250, 253]), tasks are assigned a priority during design-time, which is then used during runtime to dispatch tasks. No assumptions are made in FPS regarding *how* tasks are prioritized. Two specific priority assignment policies that can be used for FPS are *rate monotonic* (RM, [152]) and *deadline monotonic* (DM, [22]).

In RM, tasks are assigned a priority according to *rate* (or, equivalently, periods); a high rate (i.e. a short period) is translated into a high priority. Liu and Layland showed in [152] that a set of n tasks with unique priorities can always be scheduled if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$$

where C_i is the execution time and T_i is the period of task i, under the assumptions that tasks do not share resources, deadlines are equal to periods, and context switches are instantaneous. RM is *optimal* in the sense that if any static priority scheduling policy can meet all deadlines, then RM can as well. DM is similar to RM, but tasks are assigned priorities according to deadlines instead of periods [22]). DM is also optimal in the same sense as RM but this also holds when deadlines are less than periods.

A third priority assignment algorithm is *earliest deadline first* (EDF, [109, 152, 242]). Here, task instances are assigned a priority according to their deadline. EDF is optimal in the sense that if tasks characterized by arrival time, execution time and deadline can be scheduled by any algorithm, then EDF can also schedule the tasks. When deadlines equal periods, EDF is schedulable when

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Recent results in real-time scheduling theory also make it possible to combine several execution models in one system while still guaranteeing a predictable timing behavior (see, for example, [1, 2, 38, 109]). Regardless of which scheduling algorithm is used in an RTS, the timeliness of the system must be guaranteed off-line (i.e., before deployment) using *schedulability analysis*. If the system is deemed *unschedulable* (i.e., some temporal requirements cannot be guaranteed), then changes in the architecture and/or design are necessary

to guarantee proper timeliness. For offline scheduling, this process is straightforward, since a generated schedule can simply be tested for the necessary temporal requirements. However, the major hurdle in offline scheduling is to generate a feasible schedule in the first place. For approaches to generate offline schedules, see [80, 222, 280]. The schedulability analysis of online scheduled systems simply means testing whether the associated schedulability test is fulfilled.

3.2 Shared Resources and Stack Sharing

A *shared resource* is a resource that cannot safely be accessed by more than one task at the same time. Typical shared resources include memory-mapped external devices and operating system firmware or software services. Access to shared resources is usually protected using *semaphores*, *mutexes* or similar mechanisms. Mutexes can be locked and unlocked; a task that tries to lock an already locked mutex is *blocked* until the mutex is unlocked by the task that locked it in the first place. A consequence of this is that high-priority tasks may be blocked for extended time periods by low-priority tasks, a phenomenon called *priority inversion*. There exist several protocols to avoid priority inversion, the most common ones being *priority inheritance*, *priority ceiling* and *immediate inheritance*. Semaphores are similar to mutexes, but can also allow several tasks to access the same resource at once.

In the priority inheritance protocol (PIP, see [231, 232]), the priority of a task holding a shared resource is raised to the same level as a higher-prioritized task trying to access the same resource. In addition, the restriction that a task may only lock a resource for a single execution is imposed. This solves the problem of priority inversion, but may introduce deadlocks. Although deadlocks can be prevented by, e.g., imposing a total ordering of the resource accesses, blocking chains can still occur. If n is the number of lower priority tasks and m the number of distinct resources, a blocking for the duration of at most $\min(n, m)$ critical sections is possible [105], which is considered impractical in many real-time applications.

In the priority ceiling protocol (PCP, [105, 231, 232]), a *priority ceiling* is defined for each shared resource as the maximum priority of any task which may lock the resource. In PCP, the priority of a task holding a resource is raised to the priority ceiling whenever a task is blocked on the resource. The consequence of this is that a task can be blocked at most once for each resource. In addition, deadlocks are also prevented. The immediate inheritance protocol (IIP, [45]), is a simplification of PCP. In it, the priority of a task locking a

resource is immediately raised to the priority ceiling. This protocol can be implemented with relative ease, and has the advantages of having the same worst-case behavior as PCP.

3.2.1 Stack Sharing

Yet another type of resource available to embedded systems is random access memory (RAM), which is used to store temporary data during the execution of a program. There are two main types of methods for allocating memory: *static* and *dynamic* memory allocation. Static memory is allocated once, at the start of the task, and the allocated memory is freed when the task terminates. Dynamic (or *heap*) memory, on the other hand, is allocated upon request from the application, and must normally be freed explicitly by the application when the memory is not needed anymore. Failure to do so results in a memory leak, one reason why many embedded applications are restricted to only use static memory allocation.

One important part of the statically allocated memory is the *execution stack*, used to store local variables, function-call parameters and return addresses. A typical execution stack organization and some example content are illustrated in Figure 3.2. The allocation of stack space is critical in that if not enough stack space is available, a stack overflow exception will usually be raised or other data will be overwritten. Both situations may lead to a program crash or that the application does not perform as intended.

In conventional multitasking systems, each thread of execution (task) has its own allocated execution stack. In systems with a large number of tasks, a large number of stacks are therefore required. Consequently, the total amount of RAM needed for the stacks can grow exceedingly large. In order to limit the amount of RAM set aside for stack-memory in embedded systems, many RTOS:s provide means to execute multiple tasks on a single shared stack (e.g. Rubus [19], Fusion [259], Erika [85], SMX [167]). The two different task structures are shown in Figure 3.3.

Allowing tasks to share a single stack means that we must find some way of guaranteeing stack *consistency* (i.e., that the different stack areas of the tasks do not grow into each other). If we assume that a task starts using the stack as soon as it starts executing, and returns all stack space on completion, we can preserve stack consistency by ensuring that whenever a task is preempted, it does not resume execution until all tasks occupying stack space above it have completed. This is ensured in practice by not allowing tasks to suspend themselves voluntarily or to be suspended by blocking once they have started

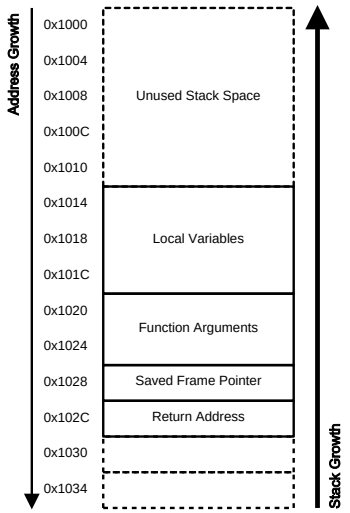


Figure 3.2: Execution stack organization and typical contents.

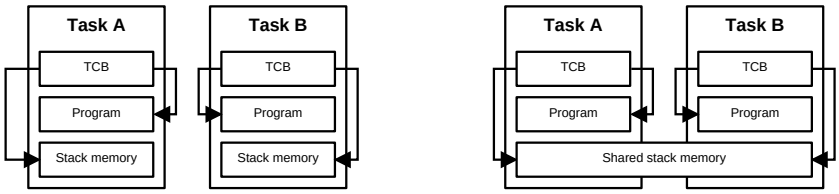


Figure 3.3: Task structure with separate stacks for each task (left) and a globally shared stack (right).

their execution. The stack resource policy (SRP), introduced in [24] by Baker (further developed in [25]) uses this principle to permit stack sharing among processes in static and in some dynamic priority preemptive systems.

In shared stack systems, one stack-frame is added to the system's stack for each level of preemption, as shown in Figure 3.4. Thus, the maximum stack usage occurs during a worst-case preemption pattern. In simple task models (commonly used in real-time scheduling theory), where tasks are assumed to be independent, any preemption pattern is possible. Therefore, we have to (pessimistically) assume that all tasks may be active and preempted at the

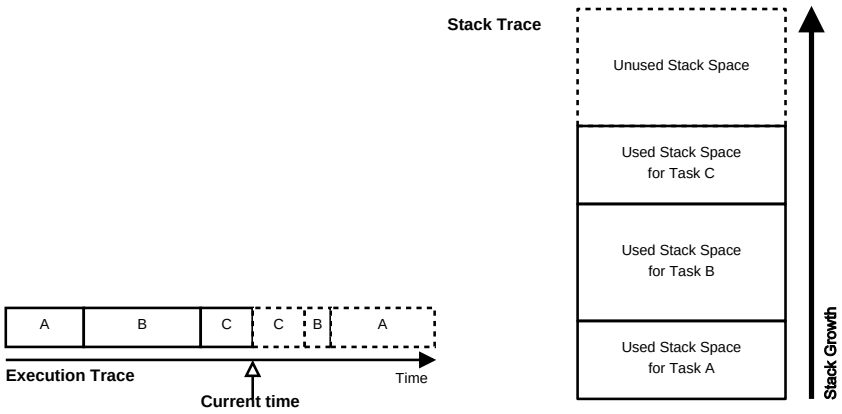


Figure 3.4: Example of execution and shared stack traces.

point where they use the most stack. The system's maximum stack-usage thus becomes $\sum S_i$ (where S_i denotes the maximum stack-usage of task i). The consequence is that in these models the benefits of using a shared stack are limited.

However, in many systems, we have information that lets us deduce that some preemption patterns are impossible. For example, in a system where multiple tasks share the same priority, no preemptions among these tasks are possible (assuming first in, first out (FIFO) scheduling within a priority level and an early-blocking resource allocation protocol such as IIP). In this case, the system's maximum stack-usage becomes

$$\sum_p \max_p(S_i),$$

where p denotes a priority level and $\max_p$ maximizes over the tasks within that priority level. If the number of priority levels is low enough, this type of analysis can provide a much lower bound on stack usage. Davis *et al.* describe this type of stack analysis and generalize it to allow non-preemption groups to be defined [63]. In Papers A and B, we develop the ideas on stack sharing further in the case where information regarding timing relations between tasks is available.

3.3 Response-Time Analysis

The response time of a task instance is the time between the invocation of the instance and the time point at which the instance finishes its execution. Response-time analysis (RTA, [20, 230]) is a family of techniques that can be used to compute the response time (RT) of the tasks in a system under different scheduling policies. The worst-case response time (WCRT) of a task is then the highest possible response time for any instance of that task. The goal of RTA is to obtain safe upper bounds on the response times of one or several tasks in a system, ideally obtaining tight WCRT estimates.

In the rest of this section, it is assumed that for all tasks, deadlines are less than or equal to the period times. The basic response-time analysis without blocking, introduced by Liu and Layland [152], defines the worst-case response time R_i of a task i as the solution to the following equation:

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.1}$$

where C_i is the WCET and T_i is the period of task i and $\mathrm{hp}(i)$ denotes the set of tasks with higher or equal priority than i. Under the assumptions that the system does not suffer from jitter and no task is blocked (i.e., no shared resources exist), RTA will yield upper-bounds on the finishing time of all tasks.

3.3.1 Extensions

In PCP or IIP, a task i can be blocked for at most one critical region by a task with lower priority. The classical RTA can be extended for the case where deadlines can be larger than the period [233], and to take blocking time, jitter and preemption delays into account [20].

In a transactional task model, tasks are divided into *transaction*. All tasks in a single transaction share one common activation event, and tasks within a single transaction may have dependencies in their release times (so-called *offsets*). In classical RTA without offsets, the critical instant for a task (i.e., the situation leading to the highest possible response time) occurs when it is released at the same time as all higher priority tasks [59]. When adding offsets, this assumption is overly pessimistic since some tasks can never be released at the same time. To tighten the analysis, Tindell [249] relaxed the notion of critical instant to mean a time point when at least one task with higher or equal priority in every transaction is released at the same time. In order to find the critical instant that maximizes the response time of the task under analysis,

an exact response-time analysis with offsets must try every possible combination of candidates among all transactions in the system, which is intractable for larger task sets. Tindell therefore provided (also in [249]) an approximate RTA with a lower time complexity. Palencia Gutiérrez and González Harbour formalized and generalized Tindell's work in [108], and in [156], Mäki-Turja and Nolin tightened the analysis and improved its run time. For tasks with offsets and jitter, Redell also presented a variant of the exact worst-case RTA in [206]. In systems where jitter is important, such as when computing end-to-end response times [108, 251] or when low jitter values are necessary to improve control performance [28], *best-case* response time analysis also becomes important to minimize jitter estimation. This is considered in work by Palencia Gutiérrez *et al.* [110] and Redell and Sanfridsson [205]. In systems where precedence information (task execution order) is available, such as in distributed systems, RTA can be pessimistic due to the infeasibility of some critical instance situations. Palencia and Harbour take precedences where each task can have at most one successor into account in [184], and Redell extends this approach to allow a task to have several successors in [204].

3.3.2 Best-Effort Response-Time Analysis

As a complement to RTA, *best-effort worst-case response-time analysis* can be employed to find an (ideally tight) *lower* WCRT bound for a task. If the best-effort worst-case response-time equals the worst-case response time upper bound, as obtained by RTA, then both the upper and the lower bounds are *exact*. However, in most cases, both the best-effort worst-case response-time estimate and the worst-case response time upper bound are inexact, and the true worst-case response time usually lies somewhere in between.

Two situations are common when trying to apply RTA to real industrial systems. First and foremost, it may be difficult to perform RTA due to, e.g., inter-task communication, which has to be taken into account. This often requires manual work and a deep understanding of how the system works. Second, RTA may return extremely pessimistic WCRT estimates due to pessimism from the WCET estimates used for task execution time, and due to the chain effects this has on the WCRT calculations.

Unfortunately, systems that exhibit behavior like this are abundant. An example of an industrial real-time system where RTA is not applicable is the control system for industrial robots, developed by ABB. This system has a very complex temporal behavior. Some tasks have execution times varying radically due to input-dependent IPC and globally shared state variables, and some tasks

may even change scheduling priority. The analytical methods' use of a task-level WCET attribute will in such cases be very pessimistic since the tasks are not independent; there are often dependencies that result in mutual exclusion between different tasks' WCET scenario.

As a result, a more detailed system model is necessary for the timing analyzis of such systems. Ideally, the model should describe the detailed execution control flow on a code level with respect to resource usage and interaction, e.g., inter-process communication, CPU time and logical resources. Methods based on measurements or simulations, have previously been shown to work well in analyzing such large and detailed models, since they only sample the system state space rather than attempting to search it exhaustively. Wegener and Grochtmann claim ([270], page 277) that

> *In practice, dynamic testing is the most important analytical method for assuring the quality of real-time systems.*

According to the same source, testing activities typically consume 50 % of the overall development effort and budget for real-time systems.

Simulation-based analysis can also be far more efficient in finding potential timing problems than system-level testing, the dominating method in industry today. Clever response time sampling can also be superior to random sampling and yield results that, together with statistical confidence estimates, can be used when few other alternatives exist. Several frameworks already exist for the timing simulation of real-time system models, e.g., the commercial tool *VirtualTime* [203] and the academic tool *ARTISST* [67]. These solutions rely on Monte Carlo simulation, which can be described as keeping the highest result from a set of randomized simulations.

An alternative method is to use metaheuristics, such as evolutionary algorithms or local search. Evolutionary algorithms have previously been tried with success in the related area of test-case generation; a review of metaheuristic search techniques for non-functional system property testing is given by Afzal *et al.* [4]. Research on test-case generation for timing analysis often focuses on measurements of execution time or analyzing systems for schedulability. For an example of the former, Wegener and Grochtmann [270] analyze WCET and best-case execution time (BCET) by measuring the execution time of a program, with input generated using an evolutionary algorithm. The method was evaluated on a number of real-time programs, and compared to Monte Carlo (random) sampling. Tasks, preemptions and communication are not considered. The results show that the evolutionary approach found more extreme execution times in all cases tried. More recently, Mueller and Wegener [271] pro-

vided a comparison of static analysis techniques and evolutionary algorithms, with regard to WCET and BCET analysis, for several real-time applications. The authors conclude that static analysis and evolutionary testing are complementary methods. Khan and Bate [134] also analyze WCET by generating test data using a genetic algorithm. The paper investigates the effectiveness of single and multi-criteria optimization for complex processors using criteria such as the number of cache and branch prediction misses and the number of loop iterations.

In the line of work which is most closely related to Paper C, evolutionary algorithms are used for verifying timing constraints in a real-time system. Here, it is common that response time is considered directly. Related work in this area is described and compared to Paper C in Section 5.1.2.

3.4 Summary

In this chapter, we provided an overview of embedded real-time systems and argued why predictability and timing requirements are essential in safety-critical applications. We also gave random-access memory (in particular, stack memory) as an example of an important resource in embedded real-time systems. We described real-time operating systems and common functionality found in an RTOS, including the scheduling of tasks, the access of shared resources and, in particular, stack sharing, in which several tasks can share a single run-time stack. We also gave a brief overview of shared stack analysis. We then continued with an overview of timing predictability in the form of response-time analysis, in which safe upper bounds on the latest finishing time of a task can be established. Finally, the chapter ended with a section on the shortcomings of RTA and alternatives in the form of response-time analysis based on measurements or simulation.

Chapter 4

Maintenance Planning

This chapter provides a review of maintenance practices for gas turbines and rail vehicles. The chapter is organized as follows. First, an introduction to the state of practice in maintenance is given. Some common maintenance policies are presented, definitions of metrics such as availability and reliability, and the practices of condition-based maintenance (CBM) are outlined. Reliability-centered maintenance is discussed next, followed by maintenance practices for gas turbines and trains are discussed. The area of maintenance optimization is then described. The chapter ends with a summary.

4.1 Introduction

In an industrial setting, breakdowns can have a significant impact on sustainability and short and long-term profitability. During a breakdown, fixed costs of equipment, real estate and labor remain constant while production is essentially zero. Therefore, rapid repair is critical to business success. Maintenance strategies define why, when and in what way maintenance is performed. There exist several types of maintenance strategies, and in practice, a mix of different strategies are almost always used. Repairing equipment after a breakdown is known as *corrective maintenance (CM)*, and is the most basic maintenance strategy; as such, it exists in some form in all manufacturing organizations. However, in many situations, the direct and indirect costs of a breakdown can be unacceptably large. This can be due to a loss of revenue or a situation where breakdown can have catastrophic consequences, such as physical injury and a substantial risk for loss of life. One example of the first is process industry

maintenance, where significant run-time is required after startup to begin producing output of sufficient quality. The goods in process at the time of breakdown, as well as the goods manufactured for a period after the breakdown, may therefore be either unusable or of less value. Breakdowns that can lead to physical injury and loss of life exist in for example automotive, aerospace and railway applications, nuclear power generation and the chemical industry.

Because of this, corrective maintenance strategies are usually complemented by breakdown avoidance strategies in a process called *preventive maintenance (PM)*. In this strategy, equipment is routinely inspected and serviced in an effort to capture developing faults early, hopefully reducing the number of breakdowns to an acceptable level. Preventive inspections also include advanced techniques for detecting invisible faults and the recording of deterioration data. The resulting time series of deterioration data can then be compared and analyzed to determine if a component has been subject to unusually heavy wear or if sudden negative trends become present. Both would indicate an imminent equipment problem. Preventive maintenance is traditionally done according to a plan specified in maintenance intervals of a suitable length and unit. For example, gas turbine maintenance is done in intervals of equivalent operational hours (EOH), equivalent number of cycles (from start to stop, EOC), and calendar time. Road and rail vehicle maintenance intervals are usually specified in either travelled distance or calendar time, or both. For gas turbines, typical interval lengths (in calendar time) range from one year and up; trains are serviced as often as once per week.

Corrective and preventive maintenance policies can be regarded as the "traditional" maintenance practices, and have been in use for decades. Unfortunately, both corrective and preventive maintenance have drawbacks. For corrective maintenance, this includes downtime in production and high maintenance costs due to secondary effects from equipment breakdown. Of great importance are also the safety and environmental issues that can be associated with malfunctioning equipment and breakdowns. However, preventive maintenance has other drawbacks, including high maintenance costs due to pessimistic maintenance intervals. Furthermore, preventive maintenance is (unfortunately) in itself a source of breakdowns due to the increased risk of human error from performing the maintenance tasks. In the end, the strategy still does not guarantee that the maintained equipment will not suffer breakdowns.

Maintenance is in general also costly, and much of it is unnecessary and avoidable. According to Wireman [278], as much as 1/3 of maintenance costs are due to bad planning, overtime costs, and limited or misuse of preventive maintenance, and are therefore unnecessary. Companies can spend as much as

its net income on maintenance [166]. Further, in [155] it is stated that maintenance expenses are usually in the range of 15–40 % of the total production cost on a yearly basis; Coetzee [53] and Bevilancqua and Braglia [33] estimate these expenses to 15–50 % and 15–70 % respectively. In [11], Alsyouf states that

> *The lack or ineffectiveness of planning and scheduling can significantly restrict the maintenance department in achieving its objectives and can thus prevent the company from maximising business profits and offering competitive advantages.*

It is reasonable to define "good" maintenance as when corrective maintenance is kept low, and as few preventive maintenance actions as possible are done [55]. Bengtsson [31] points out that fulfilling this goal demands great skill in planning proper preventive maintenance intervals and tasks. The effect on production of preventive and corrective maintenance activities is also important [43, 73]. For example, it might be beneficial to perform some maintenance activities in advance in exchange for overall higher availability, or less impact on production.

4.1.1 Maintenance Policies

A *maintenance policy* is a set of rules for how maintenance of a system should be carried out. In this section, we describe some common maintenance policies for preventive maintenance. The maintenance policies presented below are based on mathematical models using results from reliability theory and renewal theory. For a theoretical background, see books by Gertsbakh [99] (on which this section is based), Barlow and Proschan [27] and Høyland and Rausand [119].

The most common preventive maintenance policies are the *block*, *group* and *age* replacement policies. In the block replacement policy (also called *periodic replacement*), the unit is, for a period time T, preventively replaced at time instants $T, 2T, 3T, \ldots$. In addition to the PM replacements, CM is also present in that the unit is replaced at each failure which appears between preventive replacements. A variant where only operational time is considered and the goal is to maximize stationary availability is described in [99]. In a *periodic group repair* policy, a set of n units are serviced with a period of T. At service, all machines are renewed completely. No repairs are undertaken within operational periods. In an *age* replacement policy, a unit is replaced on failure or when its age reaches T, whichever occurs first. This differs from block

replacement in that age is measured relative to the previous replacement of the component. Extensions of the age replacement policy, with goal of maximizing availability where repair time is non-negligible, also exist (see, for example, [99]).

4.1.2 Availability and Reliability

The *availability* of a unit (or system) is the percentage of time the unit is available for production. Similar but slightly different is the concept of *reliability*, where downtime due to preventive maintenance is not taken into account. In power production and the oil and gas industry, availability and reliability have a great impact on plant economy. Peak periods of production are when most of the income is generated, which is why planned outages are scheduled for nonpeak periods [37] and power purchase contracts frequently have clauses on capacity payments. According to the same source, a 1 % reduction in plant availability could cost as much as $500,000 per year in loss of income on a 100 MW plant. Today, availability for small to medium size units (below 100 MW) is already between 94–97 %. For trains, availability is a very important parameter in deciding how many trains are needed for running the intended traffic. If train units have high availability, this may mean that fewer trains are needed to give an adequate service level. In [37], availability is formally defined as

$$A = 1 - \frac{PM + CM + EO}{T}$$

where PM is the amount of time spent on preventive maintenance, CM is the amount of corrective maintenance, T is the time period and EO is the equivalent outage hours due to reduced capacity. For a system generating power, EO can be defined as $EO = T_R(1 - L_A/L_D)$ where T_R is the time period of reduced load, L_A is the actual load and L_D is the desired load. Similar definitions can be constructed for vehicle maintenance, but the most common usage is to set $EO = 0$. Since availability is often a parameter in maintenance contracts, it is not common that other definitions are used. One example is to define availability as $A = 1 - M_P/T$ where M_P is time intended for production which is instead spent on maintenance. Such definitions are suitable for evaluating maintenance performance.

Since CM and EO depend on actual conditions at runtime, an optimistic assumption of no load reduction and no corrective maintenance yields the following *theoretical availability*, which is used later in Papers E and F:

$$A_T = 1 - \frac{PM}{T}$$

If the expected amount of corrective maintenance is available for a maintenance schedule, then this could easily be included in the calculations for availability when performing or after optimization. Since more maintenance is actually done when maintenance activities are grouped together, the amount of corrective maintenance would almost certainly be lower, which means a higher availability than expected.

4.1.3 Condition-Based Maintenance

Condition-based maintenance was introduced to try to avoid the pitfalls of traditional maintenance policies by maintaining the correct equipment at the right time. CBM is based on using real-time data to prioritize and optimize maintenance resources using a process of state observation called *condition monitoring*. A CBM system will ideally monitor the system continuously, acting with a preventive maintenance activity when maintenance is actually necessary. *Predictive maintenance (PdM)* takes this one step further by adding dynamic lifetime estimates and deterioration models to predict the future wear of components, allowing for production and maintenance planning in advance. In recent years, instrumentation and better tools for condition data analysis have indeed made it possible to accurately predict both the future deterioration and existing imminent failures of many components subject to physical wear. Ideally, CBM in this form will allow maintenance personnel to do only the right things, minimizing spare parts cost, system downtime and time spent on maintenance.

Although CBM in theory allows maintenance to be performed just before a potential failure becomes critical, in practice, maintenance still needs to be scheduled and planned in advance. As a direct consequence of the more exact knowledge of the maintenance need of the product, CBM maintenance "intervals" no longer remain fixed in time and easily predicted. Instead, they vary depending on the condition of several components and a variety of other factors. In [200], it is stated that in order to maximize the benefits from CBM for the enterprise, it is as important to focus on the aftermarket supply chain — i.e. the back-end of the process, including maintenance — as it is to develop better data gathering, diagnostic and prognostic techniques. Further, it is shown that optimizing the value chain results in lower costs and higher availability. Failure to plan properly for CBM, as with normal preventive and corrective maintenance, will result in availability loss due to poor grouping of maintenance, suboptimal usage of labor, and other cost inefficiencies.

4.2 Reliability-Centered Maintenance

In this section, we outline maintenance practices based on the reliability-centered maintenance (RCM) methodology [92,169,171,181,182,287]. RCM is an approach aimed at improving and optimizing maintenance practices by focusing on identifying and implementing the maintenance policies that can most efficiently manage the risk of equipment failure. In this section, we mainly use the terminology from the NAVAIR 00-25-403 management manual [171], which is consistent with the original RCM report by Nowlan and Heap [181] and compliant with the SAE JA1011 standard [218]. According to the last source, RCM addresses at least the following seven questions:

1. What is the item supposed to do, and what are its performance standards?
2. In what ways can the item fail to provide the required functionality?
3. What events cause each of these failures?
4. What are the immediate consequences to the unit when each failure occurs?
5. In what way does each failure matter?
6. Is there an activity that can be systematically performed to proactively prevent (or at least diminish to a satisfactory degree) the consequences of the failure?
7. If a suitable preventive maintenance activity cannot be found, then what should be done instead?

The initial part of the RCM process, corresponding to questions 1-5, is to identify the operating context and identify failure modes, its possible causes and effects. This is often done by performing a failure mode, effects and criticality analysis (FMECA). In answering questions 2 and 3, the failure characteristics of a physical system are defined in terms of *potential* and *functional* failures. A potential failure is "a definable and detectable condition that indicates that a functional failure will occur," while a functional failure is "the inability of an item to perform a specific function within specified limits" ([171], page 1–3). In addition, a *hidden functional failure* is defined as a functional failure undetected during normal operation.

Question 6 ("What systematic task can be performed proactively to prevent, or to diminish to a satisfactory degree, the consequences of the failure?") is

answered by setting up a set of maintenance tasks, divided into the following categories:

Servicing tasks include replenishment or replacement of consumables such as fuel, oil, filters, and anti-freezing agents, which are depleted during normal operations.

Lubrication tasks consists of the application of lubricants to specified components.

On condition tasks are periodic or continuous inspections designed to detect potential failures and therefore allow repair prior to a functional failure.

Failure finding tasks are preventive maintenance tasks performed at specified intervals to determine whether hidden failures have occurred.

Hard time tasks are the scheduled removal of an item or a restorative action at some specified maximum age.

For on condition tasks, the estimated time from a potential to a functional failure (the PF interval) is used to determine inspection intervals. Methods for estimating the PF interval include lab testing, analytical methods, in-service data evaluation and engineering judgment, which can be based on operator and maintainer input, component design knowledge and experience from different applications with similar components [171]. In reality, the true time from a potential to a functional failure is stochastic and will vary depending on the environment and operational conditions. For failure modes that can have safety or environmental effects, it is therefore important to select a PF interval that captures the situations that are possible. Note that CBM can be seen as on condition tasks with a very short inspection interval, corresponding roughly to the effective sample frequency of the condition monitoring equipment.

From a long-term scheduling perspective, there is little difference between the categories, since we, on a high-level, do not need to distinguish between what work is contained in the maintenance activity. In other words, in this thesis, a replacement (hard time task above) of a component is considered repair to a as-good-as-new state for the component, while inspections (servicing, lubrication, on-condition and failure-finding tasks) are considered “as-good-as-old” maintenance, in that we assume that such activities do not affect component lifetime. However, one significant difference between replacements and inspections is that these tasks affect each other quite substantially; since replacements restore component lifetime fully, it would serve little or no purpose to

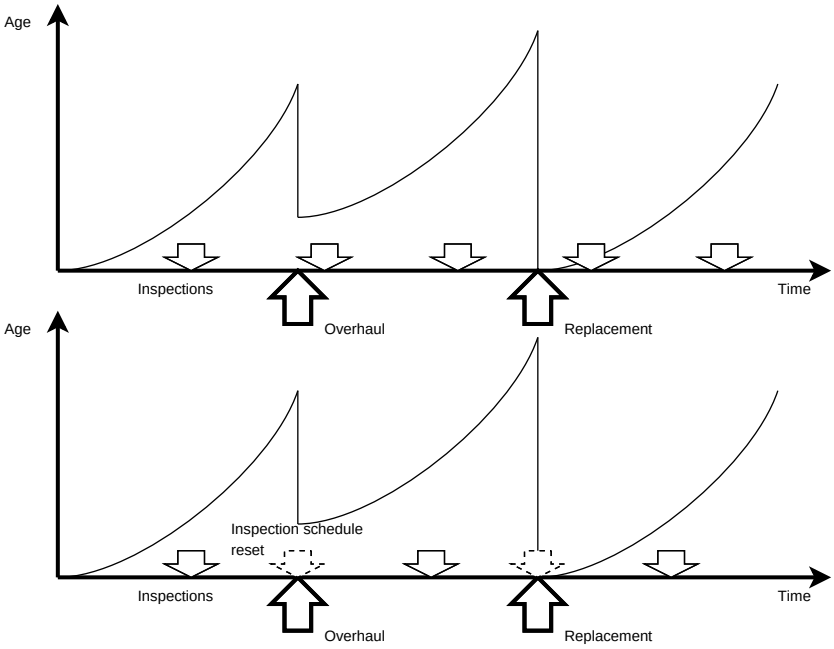


Figure 4.1: Unrelated replacements/overhauls and inspections of a single component (top), and the same situation with synchronized inspections, resulting in the elimination of unnecessary inspections (bottom).

schedule inspections independently of the performed replacements, as shown in the top of Figure 4.1. Therefore, we consider inspections of a component as dependent on the component replacements that occur during operation, as shown in the bottom part of Figure 4.1.

Once the requirements for each maintenance activity are completed, the resulting maintenance specification is *packaged* into work packages. As pointed out in [171], properly packaged preventive maintenance is more cost effective than unpackaged.. Maintenance packing is done by grouping activities that are "natural" in that they have common intervals, access the same subsystems of the unit, and/or require the same type of skills. Next, the flexibility of activity intervals needs to be determined. Safety-critical or environmentally-related activities often dictate where the groups can be performed, while economic or operational tasks can often be moved more freely. To reduce PM tracking

	Activities	
Time units	Not phased	Phased
100	1,2,3	1,2,3,5,6
200	1,2,3,4,5	1,2,3,4,7
300	1,2,3	1,2,3,5,8
400	1,2,3,4,5,6,7,8,9	1,2,3,4,9
500	*Repeat with 100 unit package*	

Table 4.1: Phasing of maintenance activities

problems, maintenance packages are also often created using multiples of a base interval.

After the initial grouping of tasks based on frequency and common factors, it may be beneficial to additionally *phase* maintenance activities in order to level the resource requirements. The following example from [171] illustrates this idea. Suppose that maintenance activities 1,2 and 3 are packaged at 100 time units, activities 4 and 5 at 200 units, and activities 6-9 at 400 units. Phasing activities essentially means spreading their occurrence over the packages to level the maintenance effort. One possible phasing of activities is shown in Table 4.1; note that under the "Phased" column, activities 6–9 are still repeated at 400 hour intervals, although in different packages from the scenario where phasing have not been performed.

The concept of "flexible packaging," where activities are packaged dynamically based on the accumulated usage or wear of the individual components, is also mentioned in [171]. According to the same source,

> *This concept allows maintenance to be performed uniquely for each end item, and therefore requires significant management oversight or facilitization using automated rulesets and tracking to ensure all maintenance is performed across the population before the RCM-derived tasks intervals. While significant operational and economic advantages are possible, the oversight required to ensure safety is not compromised should be carefully considered before adopting this approach. Development of reliable PHM*[1] *systems will make this kind of approach more easily accomplished. An additional consideration is the need to reliably predict budget, material, and resource requirements when the maintenance packages and intervals are not fixed.*

[1] Prognostics and Health Management

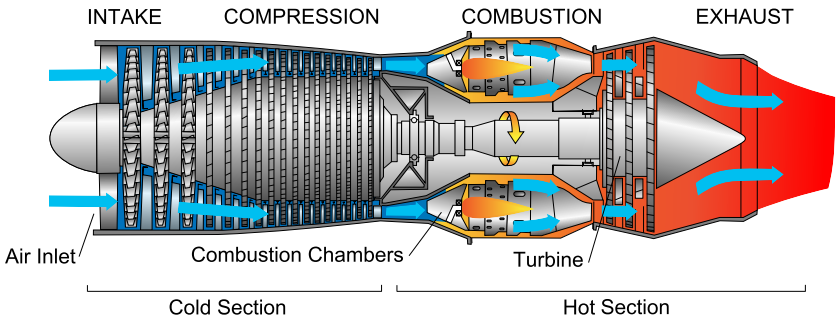


Figure 4.2: Schematics of a gas turbine. Image by Jeff Dahl, licensed under the Creative Commons License: http://creativecommons.org/licenses/by-sa/3.0/.

Papers D–F in this thesis are concerned with dynamically optimizing the packaging of maintenance. Finally, to adjust to experiences gained during operation, RCM advocates constant review and the adjustment of maintenance practices for the lifetime of the machinery.

4.3 Specific Maintenance Practices

Maintenance practices differ much between application areas, since industrial areas may have substantially different operational characteristics and demands. In this section, we will outline maintenance practices for gas turbines and train units.

4.3.1 Gas Turbines

A *gas turbine* is a rotary engine that uses the Brayton cycle to extract energy from a flow of combustion gas [37]. The archetypical example of a gas turbine is the jet engine [163], depicted in Figure 4.2. Axial flow gas turbines are typically constructed using a *compressor* (upstream), producing compressed air to a *combustion chamber*, where fuel (diesel oil, natural gas, etc.) is injected. The resulting fuel-air mixture is ignited, thereby increasing the volume and velocity (and temperature) of the gas flow, which drives the *turbine* (downstream). The turbine is coupled to the compressor, which sustains the combustion cycle. Gas turbines are found in jet aircraft, naval vessels, locomotives, battle tanks, generators and oil and gas applications. The main advantage of gas turbines is a better power-to-weight and power-to-size ratio than for piston engines.

Theoretically, a higher combustion temperature means greater efficiency, but the materials (steel alloys, nickel, ceramic, etc.) used to construct the engine parts limit the temperature at which the cycle can operate. The wear of components such as turbine blades, guide vanes, burners, and the combustion chamber itself is increased significantly with higher temperature, and considerable engineering effort is therefore spent into cooling turbine parts. One example is that the blades, guide vanes and combustion chamber are typically constructed to allow a cooling air flow to pass through the component. Although this can reduce the air flow by as much as 25 % in a modern gas turbine, cooling of this type is necessary to reduce maintenance to an acceptable level.

Turbine Maintenance

Due to its simpler construction with fewer moving parts, a gas turbine is in theory more reliable and easier to maintain than a piston engine. In practice, however, turbine components are worn heavily due to a higher working speed and temperature. In Figure 4.3, the contribution to gas turbine down time due to some major components, according to [37], is shown. According to the figure, the components in the hot parts of the gas turbine (including the first stage of the turbine) contribute 65 % of the down time of a typical gas turbine. The focus of our case study on gas turbine maintenance — presented in Papers E and F — is therefore on the hot parts, which includes the combustion chamber and blades and guide vanes in the compressor turbine. Turbine blades and guide vanes are also highly sensitive to dust, fine sand, and salts in naval environments, which works as abrasives. Therefore, air filters are fitted in environments such as deserts and on oil platforms. In some applications, filters have to be fitted and changed several times daily.

Today, the condition of the gas turbine is mainly estimated by inspection at previously planned stops. Since the gas turbine usually is in more or less constant use in between maintenance stops, the turbine cannot be inspected and/or repaired for relatively long periods of time. Therefore, methods have been developed that can not only estimate and monitor the condition and wear of the turbine during operation, but also help *predict* the future maintenance need of turbine components. The future condition of a gas turbine depends on parameters such as actual work load profile, quality of fuel, humidity, particle levels, etc. Of course, these factors have always affected the condition of the turbine, but it is not until recently that it has been possible to estimate and measure these correctly for individual components during operation. In the past, it has therefore been necessary to construct maintenance intervals from

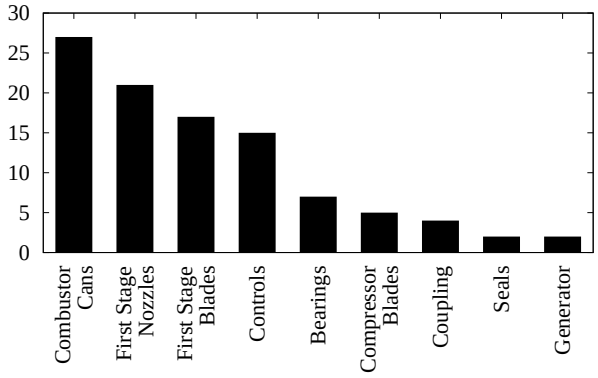


Figure 4.3: Percentage of gas turbine component contributions to down time, from [37].

the critical component (or components) that require the highest maintenance frequency. In addition, an additional worst-case scenario margin has been necessary, taking into accounts factors such as possible load variations, difference in environment, and other sources of uncertainty. These sources of pessimism present in today's maintenance intervals are natural candidates for improvement using CBM.

4.3.2 Train Maintenance

Vehicle maintenance differs in several ways from the maintenance of stationary equipment. The largest difference is that vehicles are mobile, their current and future location being dependent on the previous and future planned jobs for the vehicle. For rail vehicles, the planned jobs are usually present in the form of a timetable. Instead of having mobile repair crews visiting the site for maintenance work, the train regularly visits one or several maintenance workshops as a part of the normal duty of the train. Another difference is that the train dispatching central must make sure that the train is indeed sent to the workshop when needed. Given a timetable, the problem of allocating trains in the form of locomotives, carriages and/or multiple units to the timetabled transports is called the *rolling stock rostering problem* or the *rolling stock circulation problem* [6, 12, 86, 191, 227], and is **NP**-hard when constraints on maintenance are present [84]. We look closer on an operational version of this planning problem

when maintenance constraints are added in Paper D.

Since the freedom to plan maintenance is limited by the train rostering, the execution of maintenance actions is also limited to the time intervals when the train is actually in a workshop. These intervals may be different from the predicted time intervals since trains are dispatched according to the global train supply and the demand in the network for an operator. In addition, time-consuming setup activities are present in the shunting (movements on a rail yard) of trains to and from the workshop, and parts of the maintenance equipment might be located at other, specialized workshops in the vicinity of the main maintenance workshop. On top of this, it is frequently the case that there are several maintenance workshops located in different strategic areas of the network, often having different track layout and resource restrictions. Of importance is also the layout of the workshops, which have several resource limitations. First and foremost, a workshop contains a number of tracks for vehicles under maintenance. It is also common that tracks have different setups in the form of stationary equipment, such as lifts, graves and power lines. The current state of practice in short-term maintenance planning is, in our experience, manual planning with the aid of computerized maintenance management systems, spreadsheets and possibly project planning tools. In Paper D, we assume that the timetabled visits to the maintenance shop are planned in such a way that no workshop resource restrictions (except limited duration) apply.

4.4 Maintenance Optimization

In this section, we will give an overview of the previous work in maintenance optimization. The area of optimal maintenance and maintenance planning and scheduling has been active since the 60s, starting with the seminal work by Barlow and Hunter [26]. The book by Barlow and Proschan [27] gives a good theoretical background on reliability theory. Gertsbakh [99] also provides a good foundation of the area together with some applications.

Plenty of survey papers of the area also exist. An excellent but slightly dated overview of the many maintenance planning and scheduling applications considered is given by Dekker [69]. More up-to-date reviews are given by Budai *et al.* [43], focusing on planning models for maintenance and production, and Nicolai and Dekker [174], studying previous work in optimal maintenance of multi-component systems. Furthermore, the state of the art in applications of maintenance optimization models is discussed by Dekker and Scarf [71]. More generic mathemathical maintenance models are also reviewed by Scarf [226]. Other surveys of the area can be found in, e.g., [50, 73, 164, 193, 262, 267].

Also worth mentioning is the review of gas turbine life management by Vittal *et al.* [264], the book by Chen *et al.* [49] on machine scheduling, and the significant amount of work on maintenance management (see, for example, [34, 92, 126, 170, 183, 194, 256] and the literature review of the same area by Garg and Deshmukh [96]).

Multi-unit Maintenance

In multi-unit maintenances, the system under consideration consists of multiple units, which have identical or individual characteristics regarding failure, costs, setup activities, etc. An overview of multi-unit maintenances is given by Cho and Parlar [50]. The research papers [7, 32, 42, 66, 124, 159, 192, 198, 285] all consider multi-unit systems. In addition, the effect on the system if one unit is down is sometimes considered. In a *series system*, the system is down whenever one of the units is down; this is the model that was most suitable for the systems considered in this thesis. In a *parallel system*, the system is down if *all* units are down. A system can also be a hybrid between series and parallel. A special case hybrid is the k*-out-of-*n system model, where the system is up as long as at least k units are working.

An example of multi-unit maintenance research is given in Bris *et al.* [41, 42], who apply a genetic algorithm to the problem of minimizing perfect preventive maintenance (returning the maintained component to a state as good as new), in simulated series-parallel systems with a finite horizon. Availability requirements are considered with regard to corrective maintenance activities; preventive maintenance is assumed to be instantaneous. Sortrakul *et al.* [237] use genetic algorithms to optimize both preventive maintenance and production scheduling, which is a deterministic single-machine scheduling problem [23] with the goal of minimizing the total weighted completion time. Preventive maintenance restores the machine to a "good as new" condition, and minimal repair is assumed for corrective maintenance. Pascual *et al.* [189] consider life cycle costs when planning maintenance consisting of preventive and corrective maintenance. Other approaches, for example those made by Jayabalan and Chaudhuri [123] and Usher *et al.* [261], also consider systems under deterioration, but are not directly related to the work presented in this thesis.

Opportunistic Maintenance

The work discussed so far has in common that maintenance activities for different units have been considered more or less independent with regard to costs and duration. Of great economic importance in multi-unit systems is that the

corrective or preventive maintenance of one unit is often, due to system standstill or shared costs, an *opportunity* to perform maintenance of other units at the same time. Opportunistic maintenance (OM) is maintenance where such opportunities for more efficient maintenance exist. Opportunities can either be fixed in time and occur at specific dates, or occur due to the preventive or corrective repair of other units. However, the most common use of the term "opportunistic maintenance" is to denote models where unit failure and repair offers an opportunity for preventive maintenance of other units.

In [72], Dekker and Smeitink consider the allocation of preventive maintenance with unit duration to randomly occurring opportunities with a random duration. The authors discuss the prioritization of maintenance activities based on the component-specific cost of deviating from the optimal point of preventive maintenance. Extensions to the case when maintenance duration is non-uniform are also considered, and the authors note that for each opportunity, the optimal packing of maintenance can be decided by solving a knapsack problem [132, 160]. Galante and Passannanti [94] study the problem of deciding which components to maintain in order to guarantee a required level of reliability up to the next planned stop. The authors propose an exact cost-minimizing algorithm for this problem, and apply it to a real case regarding ship maintenance.

Maintenance with Economic Dependencies

In addition to opportunistic maintenance, there are other reasons why the joint execution of maintenance activities may be beneficial. For example, two activities may share a setup activity such as dismantling, or may be executed in parallel. Maintenance research where this is considered is sometimes called maintenance with *economic dependencies*. An overview of the area can be found in the review paper by Dekker *et al.* [73].

Several articles on maintenance models with economic dependencies have also been published. In these articles, economic dependencies are usually modeled as shared *setup costs*. The most common case is when setup costs are modeled as a constant which is independent on the clustered activities. This is the approach taken in [68, 70, 74, 273–275]. van Dijkhuizen and van Harten [263] consider a more generic dependency tree, where the leaves correspond to "basic" activities and the non-leaves (minus the root) are setup activities. Predecessor setups to an activity can therefore be shared with other activities in different branches emanating from the setups. Almgren *et al.* [9] study opportunistic replacement schedules where opportunities are possible maintenance

occasions. The duration of activities is not directly considered in the model, but shared setup costs, in the same form as in [275] and in Papers E and F, are taken into account. Setup costs may also be dependent on the calendar time of the opportunity. The problem is formulated as a MIP model, an extension of the model proposed by Dickman *et al.* [75] in that stronger constraints are used. The model is also a generalization of what was proposed in Andréasson [15, 16] by allowing time dependency. The authors shows that for each set of fixed maintenance opportunities (where setup costs are therefore in effect), the remaining problem decomposes into a linear programming problem. Therefore, binary integer variables are only needed for the opportunities. As another extension from Dickman *et al.* [75], it is shown that, for costs which are non-increasing with time, replacements will only occur at positive integer multiples of individual component deadlines.

In [245], Tan and Kramer consider opportunistic maintenance in the chemical processing industry. Monte Carlo simulation is used to estimate costs, which allows a very generic cost structure at the expense of determinism. A genetic algorithm is proposed to solve the optimization problem. The cost of production loss is considered uniform for the planning horizon, and opportunistic costs are estimated once for each component only. More complex dependencies between maintenance activities are therefore not considered. Marseguerra *et al.* [159] also apply Monte Carlo simulation and use genetic algorithms. In addition, they consider other properties such as the number of maintenance technicians available. In [288], Zhou *et al.* propose a scheduling algorithm for preventive imperfect maintenance of a multi-unit system based on dynamic programming, extending the work in [287]. Opportunities are assumed to exist whenever a component reaches its reliability threshold, and preventive maintenance is grouped using opportunistic cost savings due to downtime and maintenance costs. The downtime cost model for an activity grouped together with one at its threshold is to subtract the first cost from the total cost.

Goyal and Kusy [106] determine maintenance frequencies for a set of machines, where setup costs are constant and maintenance stops are scheduled periodically. Operating costs are assumed to increase proportionately to the length of maintenance intervals. Yamayee *et al.* [281] use dynamic programming to optimize maintenance schedules with respect to equipment reliability, demand of generating units and maintenance cost. The main difference between the work by Yamayee *et al.* and our work is that in the former, maintenance is scheduled for power-generating units on a high level. In our work, maintenance is scheduled for a single unit with the aim of obtaining maintenance packages for individual components. Since we are focused on the main-

tenance of a single unit, we also use a more detailed downtime model where small-scale effects such as resting periods are taken into account. In [275], Wildeman *et al.* discuss maintenance scheduling for a multi-component system with constant co-allocation cost savings, and where deterioration of components is also taken into account. In addition, a polynomial solution approach is presented. The polynomial solution is optimal if groups are *consecutive*, i.e., the groups are in the same order as the preferred time point of the activities. In a model with a more complex setup structure, it may be optimal to group activities non-consecutively if the earnings from doing so outweigh the costs.

Rail Vehicle Maintenance

From a system perspective, maintenance of vehicles is more complex than maintenance of stationary equipment. This is because the vehicles have to be routed to a workshop before maintenance can be performed. Therefore, research in rail vehicle maintenance often includes the associated routing problems. An exception is present in work by Hani *et al.* [111, 112] who focus on the detailed planning of work performed in the train maintenance facilities only. Cordeau *et al.* [56] give a survey of models for optimization of train routing and scheduling.

In Paper D, we approach the problem of routing vehicles to the workshop so that maintenance costs are minimized. We also consider the sub-problem of grouping maintenance activities such that the number of maintenance occasions is minimized. The problem of determining optimal vehicle routes is NP-hard in general [84], which is why we chose a heuristic method to find suitable routes.

Train maintenance routing has been considered before, and is often seen as part of the related problem of assigning trains to timetable trips. A closely related problem to the one we have considered has also been studied by Anderegg *et al.* [12], who propose a heuristic routing approach usable in a long-term perspective. Packaging of maintenance is not considered. Maróti and Kroon [161, 162] also consider the operational maintenance routing problem without considering maintenance packaging. In [161], a multi-commodity flow model is proposed to solve the problem. In [162], an integer programming formulation is presented, and a shortest path heuristic is proposed to solve the problem for a planning horizon of 1–3 days. Evaluations on a realistically-sized example show that the heuristic performs well in practice.

Sriskandarajah *et al.* [239] consider an overhaul scheduling problem for the Hong Kong Mass Transit Railway Corporation (MTRC). The routing of trains

is not considered, but workshop capacity and work force requirements are present in the model. Pěnička *et al.* [197] formalize a train maintenance routing problem and propose generation of possible changes to the traffic schedule to fulfill the model. The model is conceptually similar to the routing problem in Paper D without the sub-problem of maintenance packaging. Unfortunately, the approach is not evaluated on simulated or real data, and it is not stated whether the approach has been implemented.

4.5 Summary

This chapter contained an overview of maintenance practices with a focus on maintenance planning for rail vehicles and gas turbines. We described some common maintenance practices today, including the block, group and age replacement policies. We discussed some different definitions of availability and reliability, and gave a brief overview of condition-based and predictive maintenance. An overview of the RCM approach was also given, and then some specific issues with regard to maintenance of gas turbines and trains were described. The section ended with an overview of related work in maintenance optimization, with a focus on maintenance where there exist economic advantages in grouping maintenance. We also discussed the specific problems that arise in train maintenance.

Chapter 5

Related Work and Thesis Contributions

This chapter gives an overview of the academic and industrial contributions of the thesis, and relate our work to previous approaches to stack analysis, best-effort worst-case response-time analysis and maintenance optimization. The chapter is organized as follows. In the first section, a summary of the academic contributions is given, followed by a more detailed description of the contributions within each of the three areas. For each area, a comparison with related work and a summary of the academic contributions of the corresponding included papers are given. The industrial impact of the thesis is then described, followed by a list of the included publications together with a description of the author's role. Also included is a list of other publications by the author. The chapter ends with a discussion on future work.

5.1 Academic Contributions

The academic contributions of this thesis can be summarized as follows:

- In the area of stack analysis, we give several new efficient algorithms for analysis of shared-stack usage, and compare the algorithms to previous approaches.
- In the area of best-effort response-time analysis, we give a new efficient hill-climbing algorithm with random restarts for the problem of estimating the highest response-time in a complex system, and evaluate and

compare the algorithm with a previous evolutionary algorithm.

- In the area of maintenance planning, we give several new methods for dynamic planning of train and gas turbine maintenance, and compare the results to the state-of-practice.

In the rest of this section, we briefly describe the differences between Papers A–F and related work in the three areas of shared stack analysis, best-effort response-time analysis and dynamic maintenance planning. For each area we also outline the academic contributions of the included papers.

5.1.1 Shared Stack Analysis

The term stack sharing is commonly used to describe the ability to utilize either a common run-time stack or a pool of run-time stacks. Stack sharing in the SMX RTOS [167] is an example of the latter, where released tasks fetch a stack from the pool of available stack areas, returning it on termination. A different approach is proposed by Middha *et al.* [168], where the stack of a task is allowed to grow into the stack area of another task. However, the most common type of stack sharing seems to have evolved from Baker [24, 25], where the proposed *stack resource policy (SRP)* allows tasks to share a single run-time stack. Stack consistency is achieved by not allowing preempted tasks to resume until all tasks occupying stack space above it have finished. SRP permits stack sharing among processes in static and in some dynamic priority preemptive systems. This type of stack sharing can be efficiently implemented in systems where tasks have run-to-completion semantics and do not suspend themselves, and is supported by several commercial real-time operating systems, e.g. RTA-OS [153], Rubus OS [19] and Fusion RTOS [259]. In Papers A and B, we use this notion of stack sharing, and assume that several tasks use one common, statically allocated, run-time stack.

Since in SRP, a preempted task is not allowed to resume until the tasks occupying the stack space above it have terminated, the possible preemptions between tasks become crucial in determining the maximum possible stack memory usage. The basic method to determine this in SRP and similar policies is to identify the maximum stack usage for the tasks on a single priority level (or preemption level). Since tasks on the same priority level cannot preempt each other, the sum of these maximums for all priority levels then constitutes a safe upper bound on the total stack usage.

Several authors have also addressed the minimization of stack space allocation. A common way of reducing the number of possible preemptions (thereby

also reducing stack requirements), is to allow tasks to disable preemptions from tasks up to a specified priority, the so-called *preemption threshold*. Tasks with a higher priority than the threshold are still allowed to preempt. Wang and Saksena [268] address the problem of determining an optimal priority and preemption threshold for a given task set. However, due to a potentially large search space, the branch and bound algorithm presented is not very efficient. In [220] Saksena and Wang revisit the efficiency problem of the algorithm in [268] and present three algorithms with different computational complexity. Gai *et al.* [93] introduce SRP with preemption thresholds (SRPT) and give a procedure that can minimize shared-stack usage without jeopardizing schedulability. The procedure achieves this by using non-preemption groups for tasks using SRPT, and extends [220] by taking the stack usage of tasks into account. Ghattas and Dean [100] also investigate stack space requirements under preemption threshold scheduling. Davis *et al.* [63] also address stack memory requirements by using non-preemption groups to reduce the amount of memory needed for a shared stack. It is shown that the number of preemption levels required for typical systems can be relatively small, while still maintaining schedulability.

Although non-preemption groups and preemption thresholds can reduce the amount of RAM needed for a shared stack, the use of these affects a system by restricting the occurrences of preemptions, which can have a negative effect on schedulability. Furthermore, the method we present in this paper can be applied after preemption groups have been assigned, thereby reducing the system stack further.

To obtain an upper bound on stack memory usage for a given task, the possible control-flow paths of the task within an application must be analyzed [116]. Bounds on maximum stack usage of a given task can be found by abstract interpretation of an application with tools such as AbsInt [3] and Bound-T [248]. Chatterjee *et al.* [48] study stack boundedness for interrupt-driven programs. The programs are modeled using the interrupt calculus of Palsberg and Ma [185]. In [208] Regehr *et al.* present a method to guarantee stack safety of interrupt-driven software. The method works by computing the worst-case memory requirements of individual interrupt handlers, and by then performing preemption analysis between handlers.

A large number of publications also address preemption analysis, see, e.g. [13, 51, 77, 147, 202, 207, 243]. For example, in [147] Lee *et al.* present a technique to bound cache-related preemption delays in fixed-priority preemptive systems. Our work relates to theirs in that we also investigate nested preemption patterns. However, our objectives differ in that Lee *et al.* focus on timing

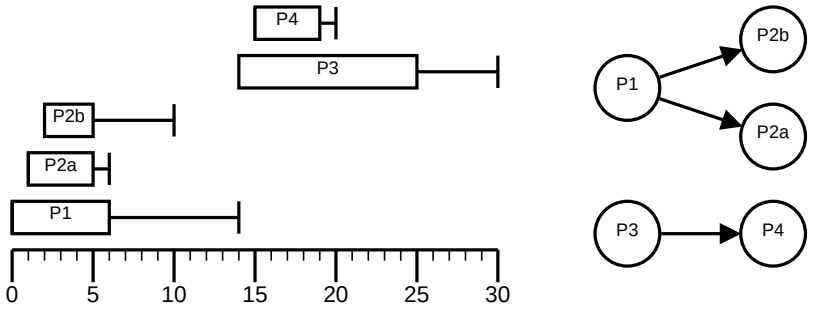


Figure 5.1: Offset relations (left) and the resulting preemption graph (right) for a set of tasks with priority 1–4.

delays caused by cache reloading and preemption patterns, whereas we address shared memory requirements.

Contributions

In the previous section, we saw that the most common analysis technique for stack sharing is to use the sum of the stack usage for each preemption level. However, in some cases, more information that can be used for computing a safe stack upper bound exists. One source of available information is the timing relations between tasks. For example, data regarding the earliest possible activation time point, a so-called *offset*, can together with response time data aid in deciding which preemption patterns are possible. The situation is illustrated in Figure 5.1, showing five tasks of different priority. The tasks are also affected by a short high-priority service interrupt which is taken into account in the response times of the tasks. Tasks are named P1–P4, with a high number indicating a high priority; note that P2a and P2b share the same priority. The release time points and latest finishing time points of the task set, assumed to be schedulable, are shown. We assume a maximum stack usage of 1 for each task. The relations (due to offsets and response times) between tasks make it possible to deduce that none of the tasks P1, P2a and P2b can be preempted by any of tasks P3 and P4, since they will never execute simultaneously. Furthermore, tasks P2a and P2b share the same priority and therefore may not preempt each other.

The information on possible and impossible preemptions can be collected in the form of a *preemption graph*, shown in the rightmost part of Figure 5.1. In

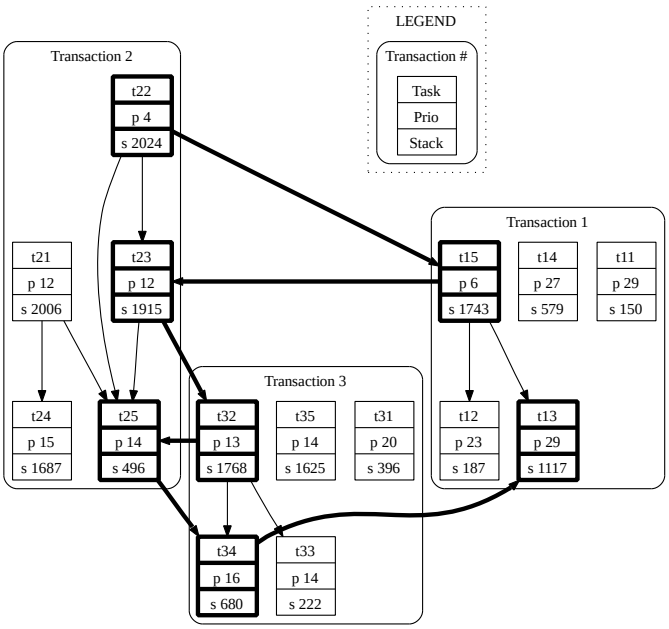


Figure 5.2: Example of a maximum stack utilization preemption chain in a system with three transactions and 15 tasks in total. Thin arcs indicate possible intra-transaction preemptions between tasks due to increasing priority and offset relations The thick arcs indicate the longest path in the maximum possible preemption chain. Note that of the possible inter-transaction preemptions, only the ones in the longest path are shown.

this figure, an arc between two tasks indicates that the tasks may be preempted in the order of the arc. From this graph, we can deduce that we will need at most 2 units of stack space, a significant reduction from the estimate of 4 obtained from using the sum of the maximum stack usage on each level.

Paper A develops this idea further for systems where tasks that share the same stack have an offset relation. However, in some systems, groups of tasks (called *transactions*) share a single activation event. The different activation events in those systems are independent. Therefore, preemptions that are impossible due to offsets and response times can only be accurately taken into account within transactions. In Paper B, the ideas from Paper A are extended to handle this more generic system model, and to further tighten the analysis us-

ing non-preemption relations from other sources, for example shared resources or precedence relations. A small example of a maximum stack utilization preemption pattern is shown in Figure 5.2.

In more detail, the contributions are the following:

- A general and exact formulation of the maximum stack usage problem, which is applicable for any preemptive system model based on dynamic (run-time) properties.
- Several novel methods to determine the maximum stack memory used in preemptive, shared stack, real-time systems. By approximating the run-time properties, together with information about the underlying run-time system, these methods can safely approximate the maximum system stack usage at compile time. The thesis also contains proofs of correctness of the given algorithms.
- Two comprehensive simulation studies where we have evaluated our techniques and compared them to the traditional methods to estimate stack usage. We found that our methods significantly reduced the amount of stack memory needed.

5.1.2 Best-effort Response-time Analysis

Recall that the response time of a task is the time taken from task invocation to termination. Although analysis of response-time with regard to other aspects than the worst-case response-time, for example average response-time can also be interesting, we will focus on worst-case response time. Worst-case response-time analysis includes standard approaches such as RTA [125, 152] and formal analysis tools like UPPAAL [29, 260], which can also be used for this purpose. However, the state space for industrial-sized models can grow too large for formal analysis tools to be practically useful.

An alternative method is to use metaheuristics such as genetic (or *evolutionary*) algorithms [103]. In Section 3.3.2, we described related work with regard to metaheuristics for execution-time analysis. However, the line of work most closely related to Paper C is the use of evolutionary algorithms for verifying timing constraints in a real-time system. In this line of work, response time is often considered directly, since it is more appropriate in a system-wide analysis than execution time is. As an example, Alander *et al.* [5] use genetic algorithms to generate test cases for a software relay system used in electrical networks. The purpose of the genetic algorithm is to provoke high response

times for the software, which executes as a single task in a simulation environment. Preemptions and communication between tasks are therefore not considered. Another approach is given by Briand *et al.* [39, 40], who investigate using genetic algorithms for stress-testing real-time systems in the sense that test cases that maximize the chances of deadline misses are constructed. The genetic algorithm operates on a sequence of release times for aperiodic tasks. Input data is not considered, since this is considered estimated for in the tasks' WCET.

A problem with evolutionary algorithms is that they are not particularly suitable to guaranteeing a high testing coverage. Tlili *et al.* [254] extend the basic evolutionary testing approach by seeding the algorithm with test data to achieve a higher structural coverage. Experiments show an increased reliability in the results and an increased efficiency in terms of generations needed.

In a distributed system, response time can involve communication over several distributed nodes, and timing analysis is therefore more complex than for single nodes. Samii *et al.* [221] aim to find extreme response times for distributed systems by optimizing a set of simulation parameters for models containing temporal attributes and communication. A genetic algorithm is used to explore combinations of task execution times in order to maximize end-to-end response time. The flow of control within tasks is not considered. Their results depend on the method developed by Racu and Ernst [199] for identifying situations where decreased execution times can lead to increased response times. Also worth mentioning is the analysis framework by Kim *et al.* [135], which has a similar basis in the use of temporal task attributes.

Kraft *et al.* [141] present a meta-heuristic approach for best-effort response-time analysis of models of complex legacy systems using ideas from genetic algorithms. The approach is based on a simulator using a schedule of random number generator seeds, in turn used to generate random numbers for the parameters of the adhering system model. The seed of the random number generator can be changed at arbitrary time points, and thus provides a form of control mechanism.

The work presented in Paper C is an extension of [141], introducing an explicit representation of input data that is more suitable for the analysis undertaken. Contrary to previous approaches, we use the well-known, but in this area rarely used, hill-climbing algorithm for response-time analysis, and we take into account system-level properties such as preemptions and task communication. To the best of our knowledge, this approach at response-time analysis has not been tried before. Results are promising in that convergence for small systems is very quick, and the less complex algorithm performed, in all

cases tried, better than the genetic algorithm we used for comparison..

Wegener and Grochtmann [270] use evolutionary algorithms for finding extreme WCET and BCET estimates. They report that local search, such as hill climbing and simulated annealing, had little effect in improving the results of the evolutionary algorithm, and suggest that this effect comes from already having reached a fitness plateau or local minima by using the evolutionary algorithm. The authors then state:

> *The optimal solution sought represents an isolated and small subdomain and is best found by sampling the input domain widely.*

We believe that this conclusion is also valid for estimating response time. It is likely that the results obtained in Paper C can partially be accounted to the randomization and iterative restart of the hill-climbing algorithm, which helps in exploring plateaus and escape local minima.

Contributions

Paper C proposes an complementary approach to traditional RTA, in contrast can be applied to a wide range of complex industrial systems, but does not guarantee that the produced estimates are upper bounds on the WCRT. The approach is based on simulation optimization, which is capable of reproducing the application behavior that causes a specific response time. The main merit of the proposed approach is that it can be used for testing purposes: showing that the response time of a task *exceeds* the task deadline is enough to deem the system unschedulable, and therefore unsafe. More specifically, Paper C contains the following contributions:

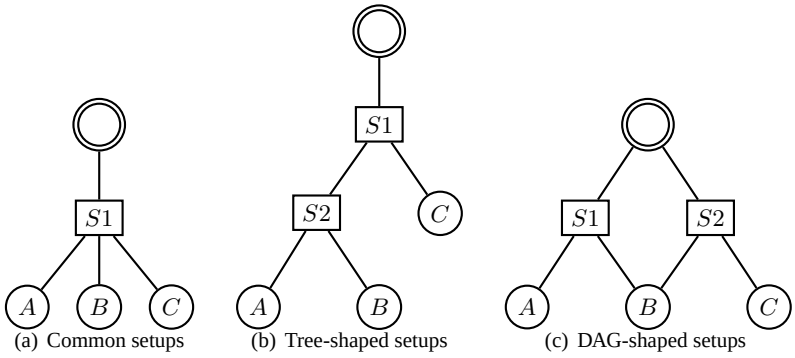
- An explicit representation of simulation instances in the form of inputs such as execution time, arrival jitter and external input stimulus has been defined.
- A novel algorithm for manipulating simulation parameters, based on the well-known idea of hill-climbing with random restarts.
- A thorough experimental evaluation of performance, scaling and convergence of the new algorithm, comparing the results to those obtained from using a genetic algorithm and Monte Carlo simulation. In the evaluation, we show that the new algorithm is significantly better than previous approaches in identifying extreme response times using a limited number of simulations.

5.1.3 Maintenance Planning and Scheduling

In this section, we compare the work in Papers D–F with what has previously been done in the area of maintenance planning. The body of research aimed at the railway sector is significantly smaller than for the more generic problem of maintenance planning with economic dependencies. The main differences between Paper D and the previous work in the railway domain is that in the former, we study stochastic maintenance predictions in the form of a Gaussian distribution, where the risk level of overrunning a subsystem counter is the basis for setting a maintenance deadline in a global unit, for example distance. We also include the risk levels and the change in safe lifetime estimates occurring during condition monitoring as the input to the planning problem. This problem in turn is composed of two subproblems; first, finding suitable circulation plans so that the maintenance cost is minimized, and second, the packing of maintenance on a component level as a subproblem to compute the global maintenance cost. As far as we know, this approach has not been tried before in the railway domain. However, it should be pointed out that the approach presented in Paper D can benefit from more advanced models and solution approaches for both the railway circulation problem and for the maintenance planning subproblem. This is discussed in more detail in Section 5.5.3.

The previous work most closely related to Papers E and F is concerned with maintenance scheduling with economic dependencies between maintenance activities. As an example, two activities may share the same dismantling activity, or may be performed in parallel. Our work and previous approaches differ mainly in the economic effect of grouping activities. The situation is illustrated in Figure 5.3. Most articles on maintenance models with economic dependencies consider a common setup cost for all maintenance activities performed at a single maintenance occasion. The approach is illustrated in Figure 5.3(a), where activities A, B and C all share a common setup cost, $S1$. The setup cost is used to represent the actual cost of dismantling and preparation work before, during and after the maintenance occasion. In most cases, the setup cost is also constant and therefore independent on the date of the maintenance occasion. This is the approach taken in [68, 70, 74, 273–275]. In some other papers, the cost is allowed to vary with the maintenance occasion date, see for example the work by Almgren *et al.* [8, 9].

Common costs are suitable for the modeling of costs due to shared activities, for example where there is a single shared cost associated with taking down the system [8, 244, 279], and where the setup cost is independent on the activities performed during the opportunity. This is true for many practical



(a) Common setups (b) Tree-shaped setups (c) DAG-shaped setups

Figure 5.3: Economic dependencies with a common setup cost (a), with tree-shaped setups [263] (b) and with DAG-shaped setups [8] (c).

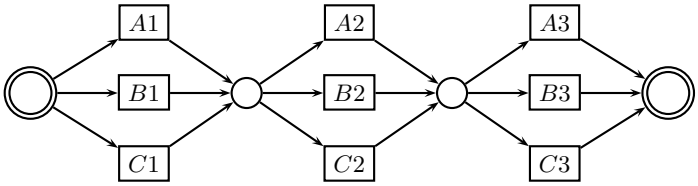


Figure 5.4: Economic dependencies due to parallel-time duration computations.

applications, which is why we include the same type of time-dependent common setup cost in Papers E and F. The model is however less suitable when the actual activities carried out during the opportunity have more intricate dependencies, and when the duration of the stop has a significant effect on the total cost. The single work most closely related to Papers E and F is due to van Dijkhuizen and van Harten [263], who consider shared setup costs and dependencies that form a tree. The situation is illustrated in Figure 5.3(b), where setup S2 is shared between activities A and B, and setup $S1$ is shared between setup $S1$ and activity C. If in this example activity A or B were planned for a single stop, then both $S1$ and $S2$ would have to be carried out. However, if only activity C was to be carried out, then only setup $S1$ would have to be performed. The tree breakdown of dependence is attractive since it corresponds more closely to the assembly structure of many units (see, e.g., the paper by Sculli and Suraweera [228]), and is therefore suitable to represent setup costs

due to the disassembly of a unit.

However, it is worth noting that a tree structure does not allow the modeling of situations like the one in Figure 5.3(c), where setup $S1$ is shared between activities A and B, while setup $S2$ is shared between activity B and C. Almgren *et al.* [8] give an informal example of this type of more complex dependencies between setups in the area of jet engine maintenance, where there are several possible ways to disassemble and reassemble the engine. In this case, only one disassembly path in Figure 5.3(c) needs to be taken to reach a single component. Since an industrial gas turbine and a jet engine are similar in construction, the example in [8] is also relevant for our application. However, one significant difference is that a jet engine is usually replaced as a unit and then serviced offline. Gas turbines used for the application we consider are often serviced on site, and the main cost driver is the downtime of the gas turbine.

In Papers E and F, we therefore opted for a downtime-dependent cost model more accurately representing the loss-of-production costs. For this purpose, we use a detailed but manageable model of economic dependencies due to the effect of parallel work on downtime. In this model, illustrated in Figure 5.4, work in an activity is divided into different global *phases*. Typical phases include the shutdown of the turbine, one or several maintenance phases, a startup phase, and a testing phase. All jobs (at a single maintenance occasion) which are performed within a phase can be done in parallel, but the phases have to be performed in series. Activities which cannot be done in parallel should therefore be separated into different phases. After computing the work time for the activities at an opportunity, we can then proceed to compute an additional night and weekend rest time, as shown in Paper F.

As an example, the work-time model illustrated in Figure 5.4 contains phases 1–3. In the figure, maintenance of type A consists of the activities $A1$, $A2$ and $A3$, B consists of $B1, B2$ and $B3$ and C consists of $C1, C2$ and $C3$. The activities $\{A1, B1, C1\}$, $\{A2, B2, C2\}$ and $\{A3, B3, C3\}$ can be done in parallel. Each activity has a duration (except the leftmost and rightmost nodes which are used to indicate the start and finish of the maintenance occasion). Therefore, the work time duration u_j (not considering night and weekend rest) for a maintenance occasion j can be computed as the longest path in the graph, or equivalently as

$$u_j = \max\{A1, B1, C1\} + \max\{A2, B2, C2\} + \max\{A3, B3, C3\}.$$

Given a work time duration u_j, a downtime cost l_j for the occasion j and a function $\mathcal{D}$ (given in Paper F) adding resting time, we can then compute the cost of a single occasion j due to downtime as $lpcj\mathcal{D}(u_j)$. Note that the cost of

downtime is unique for each occasion. This is important since in practice, the downtime cost varies with the price of the produced commodity and the cost of operation. For the same reason, there might also be periods of downtime that are essentially free with regard to downtime cost. Such periods occur when the plant is down due to external circumstances not captured by the maintenance model. Examples include the maintenance of other systems not considered within the application, and downtime due to weekends or vacation periods. As far as we know, the approach outlined above has not been tried before in the field of maintenance planning.

Contributions

In Papers D–F, we have studied the problem of preventive maintenance planning and scheduling under condition monitoring. In traditional maintenance planning, preventive maintenance is statically scheduled at design-time. In condition-based maintenance, preventive (and ideally corrective) maintenance must continuously be rescheduled (and therefore re-planned) to take full advantage of potential cost savings. In essence, the problem is to dynamically decide which maintenance activities should be grouped together at which point in time. For rail vehicles, this includes the routing of vehicles to the maintenance workshop. We have developed a simulation environment for stochastic predictive maintenance where vehicles are dynamically routed and maintenance is planned into packages. We have also developed a tool for the heuristic optimization of preventive maintenance stops. In detail, the contributions of this thesis are the following:

- A methodology for dynamic planning of the maintenance of trains, in which trains are rerouted to maintenance shops according to dynamically changing maintenance deadline estimates. Maintenance is packaged and planned according to the vehicle occupancy in the workshop.
- A precise definition of the maintenance scheduling problem with opportunities, which allows maintenance to be planned with regard to both loss-of-production costs and direct maintenance costs. We also prove that the scheduling problem is **NP**-complete.
- An implementation of a heuristic algorithm that can quickly solve the problem for practical purposes. We also describe the implementation and the deployment of the scheduling tool, PMOPT.
- An evaluation of the results of maintenance scheduling on four variants

of a real-world scenario, and a comparison of the results of our algorithm to the results from using mixed integer linear programming.

5.2 Industrial Impact

The work presented in this thesis has had substantial impact in industry. All included papers have either been implemented and deployed in industry, or have been evaluated on data from real-world industrial applications. In detail, the industrial impact has been the following:

- The polynomial-time stack analysis algorithm that is presented in Paper A (which has also been further developed in Paper B) was developed with the goal of being applicable for the Rubus OS from Arcticus Systems [19]. In particular, the hybrid system model in Rubus OS consists of one set of time-triggered offline-scheduled tasks, together with event-triggered tasks in the form of higher-priority interrupt tasks and lower-priority soft real-time tasks. The statically scheduled tasks can share a single execution stack, and can be considered a single transaction. The stack analysis from Paper A with polynomial complexity was therefore chosed for deployment [117], and is included in the Rubus ICE development environment.

- The best-effort response-time analysis method presented in Paper C has been applied to models of industrial systems, but has not yet seen industrial deployment. However, it seems likely that the developed method could be used to estimate the worst-case response-time for a real industrial system. The simulator RTSSim is generic and can be used to simulate the behavior of most commercial RTOS:s. However, for the method to be useful for large-scale complex real-time systems, a model extraction tool should be employed to simplify the simulated program and decrease simulation time. Such a tool (MXTC, Model eXtraction Tool for C) is currently in development along the lines of the approach proposed in [14].

- The approach for combined maintenance routing and planning proposed in Paper D have been developed in collaboration with Bombardier Transportation AB, and has been evaluated on real timetable data and train circulation plans. The interest from the railway industry has been significant, and we have frequently shown the demonstrator application in relevant practical fora. However, the approach has not yet been deployed

in practice. One possible explanation for this is that the rail industry, having a history spanning 150 years back, is slightly more conservative than other industrial areas. In addition, there has traditionally been less competition in the rail industry than in other industrial areas due to a state-owned railway system. The relatively recent deregulation of the Swedish railway sector has changed this picture substantially, but has also divided maintenance responsibility between several actors. There is also still a lack of competition between operators, which have not worked in favor of introducing new technology. However, the plans to further deregulate the railway sector and allow competing operators on equal terms will likely open up for approaches such as the one proposed in Paper D.

Furthermore, the maintenance packaging heuristic from Paper D was the basis for later development into the maintenance optimizer used in Papers E–F. The optimization algorithms employed in the latter papers have been implemented and integrated in a tool, PMOPT, which has been deployed and is currently in use at Siemens Industrial Turbomachinery AB (SIT AB). The development effort started with an extension of the approach proposed in Paper D, and was then further developed in close collaboration with SIT AB. The development and deployment effort is described in Paper F. PMOPT has now been running operatively for almost a year within two maintenance contracts. in the first, PMOPT is fully operational, while in the second, PMOPT is used for validation and testing purposes of the full CBM strategy. Testing is done mainly for gaining feedback from practical experience, monitoring of environmental variables and time increments. Within a few years, four or five people working within maintenance planning are expected to use the tools for 10–15 different operational contracts. The estimated availability improvement of using PMOPT is 0.5–1.0 %, a substantial increase considering that availability is currently in the range of 97–98 %. This amounts to a decrease in downtime of 16–50 %.

5.3 Publications Included in the Thesis

This section lists the papers included in this thesis (including bibliographical data) and details my contribution to each of them.

- K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin. Determining maximum stack usage in preemptive shared stack systems. In Proceedings of the 27th IEEE Real-Time Systems Symposium, Decem-

ber 2006.

I was one of the authors of the paper. My main contributions are the construction of the algorithm and the proofs of safety and correctness.

- M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin. Bounding shared-stack usage in systems with offsets and precedences. In Proceedings of the 20th Euromicro Conference on Real-Time Systems, July 2008.

  I was the main author of the paper and coordinated the work. I constructed both algorithms and most of the proofs with some help and checking from the other authors. I also implemented the algorithms and helped in performing the evaluations.

- M. Bohlin, Y. Lu, J. Kraft, P. Kreuger and T. Nolte. Best-Effort Simulation-Based Timing Analysis using Hill-Climbing with Random Restarts. In Proceedings of the 15th International Conference on Real-time and Embedded Computing Systems and Applications, August 2009.

  I was the main author of the paper and coordinated the work. I constructed and implemented the optimization algorithm.

- M. Bohlin, M. Forsgren, A. Holst, B. Levin, M. Aronsson, and R. Steinert. Reducing vehicle maintenance using condition monitoring and dynamic planning. In Proceedings of the 4th IET International Conference on Railway Condition Monitoring, June 2008.

  I was the main author of the paper and coordinated the work. I implemented the heuristic optimization algorithms and performed the evaluation.

- M. Bohlin, M. Wärja, A. Holst, P. Slottner, and K. Doganay. Optimization of condition-based maintenance for industrial gas turbines: Requirements and results. In Proceedings of ASME Turbo Expo 2009: Power for Land, Sea and Air, paper number GT2009-59935, Orlando, Florida, USA, June 2009.

  I was the main author of the paper and coordinated the work. I performed the experiments and designed and implemented the optimization software.

- M. Bohlin, K. Doganay, and P. Kreuger. Scheduling gas turbine maintenance based on condition data. In Proceedings of the 21st Innovative

Applications of Artificial Intelligence Conference, Pasadena, California, USA, July 2009.

I was the main author of the paper and coordinated the work. I designed and implemented the optimization software and set up and performed the experiments with help regarding schedule feasibility from M. Wärja at Siemens Industrial Turbomachinery AB.

5.4 Relevant Publications Not Included in the Thesis

This section lists related papers authored and co-authored by the thesis author that are not included in the thesis.

- M. Bohlin. Composing global constraints for local search. In Proceedings of 15th International Conference on Applications of Declarative Programming and Knowledge Management, Fraunhofer FIRST, Berlin, and University of Potsdam, 2004.
- M. Bohlin. Design and implementation of a graph-based constraint model for local search. Licentiate thesis, April 2004.
- M. Bohlin, K. Hänninen, and J. Mäki-Turja. Shared stack analysis in transaction-based systems. In J. Hansson, editor, Work in Progress Proceedings of the IEEE Real-Time Systems Symposium, pages 37-40, December 2007.
- M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin. Safe shared stack bounds in systems with offsets and precedences. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-221/2008-1-SE, Mälardalen University, January 2008.
- M. Bohlin, P. Kreuger, M. Aronsson, and Malin Forsgren. Ansatser för flexibel planering och schemaläggning av tågtidtabeller (In Swedish). Technical Report T2006:08, SICS (Swedish Institute of Computer Science), 2006. ISSN 1100-3154.
- M. Bohlin, Waldemar Kocjan, and P. Kreuger. Designing global scheduling constraints for local search: A generic approach. Technical Report T2002-20, Mälardalen University, November 2002.
- M. Bohlin. Constraint satisfaction by local search. Technical Report T2002:07, Mälardalen University, June 2002.

- M. Bohlin. Improving cost calculations for global constraints in local search. In Pascal Van Hentenryck, editor, Principles and Practice of Constraint Programming, page 772. Springer, September 2002.
- J. Ekman, A. Holst, M. Aronsson, M. Bohlin, M. Forsgren, and S. Larsen. Time - en gemensam informationsutbytesplattform för järnvägstransportbranschen (In Swedish). Technical Report T2006:03, SICS (Swedish Institute of Computer Science), 2006. ISSN 1100-3154.
- K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin. Analysing stack usage in preemptive shared stack systems. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, Mälardalen University, July 2006.
- K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin. Determining maximum stack usage in preemptive shared stack systems. In Proceedings of the 9th Real-Time in Sweden Conference, pages 118-126, August 2007.
- P. Kreuger, M. Aronsson, and M. Bohlin. Leveranstågplan: specifikation och åtagande (In Swedish). Technical Report T2006:02, SICS (Swedish Institute of Computer Science), 2006. ISSN 1100-3154.
- Yue Lu, M. Bohlin, J. Kraft, P. Kreuger, T. Nolte, and C. Norström. Approximate timing analysis of complex legacy real-time systems using simulation optimization. In Proceedings of the Work-In-Progress track of the 29th IEEE Real-Time Systems Symposium, December 2008.
- B. Levin, A. Holst, M. Bohlin, R. Steinert, and M. Aronsson. Dynamic maintenance. In Proceedings of the 21st International Congress and Exhibition on Condition Monitoring and Diagnostic Engineering Management, June 2008.
- M. Wärja, P. Slottner, and M. Bohlin. Customer adapted maintenance plan (CAMP), a process for optimization of gas turbine maintenance. In Proceedings of the ASME Turbo Expo 2008 Gas Turbine Technical Congress and Exposition, June 2008.
- M. Aronsson, M. Bohlin and P. Kreuger. MILP formulations of cumulative constraints for railway scheduling — A comparative study. To appear in Proceedings of the 9th Workshop on Algorithmic Methods and Models for Optimization of Railways, 2009.

5.5 Future Work

The work in this thesis could be developed further in several ways. In this section, we outline some possible future research directions.

5.5.1 Stack Analysis

- The step from the *analysis* of stack requirements to the *design optimization* of real-time systems with the goal of minimizing stack requirements is quite small, and is therefore a prime candidate for future research.

- The stack analysis of real systems still require that the maximum stack usage per task is computed. This problem is in itself complex, and has previously been solved using abstract interpretation or other analysis methods. An alternative would be to analyze the shared stack in a best-effort manner, similar to what we propose for response-time analysis in Paper C.

5.5.2 Best-effort Response-time Analysis

- Automatic or semi-automatic model extraction is a prerequisite for the industrial deployment of the best-effort response-time analysis methods presented in Paper C. This work is ongoing (see, for example, [118, 139, 140]), but further research efforts are needed here to demonstrate the usefulness of the proposed approaches.

- The validity of the analyzed model with regard to temporal properties is crucial in order to obtain any confidence in the response-time results. Here, model validation [224] could be employed to ensure that the model is an accurate representation of the modeled system.

- The optimization method in itself can be improved in many ways. For example, in some applications, a correlation seems to exist between high (or low) input values for certain inputs and high response-time for the task under analysis. The algorithm could be modified to gather statistics regarding the existence of such relations. It could also be improved by placing more focus (in a intensification phase) on selecting values that have previously yielded high response-times.

- The lower-bound on worst-case response time that the method yields cannot be used in safety-critical applications without some measure of

accuracy. Statistical estimates of the worst-case response time could be useful for this purpose.

5.5.3 Maintenance Scheduling

Outlined below are some possible ways in which the maintenance planning concept presented in Papers D–F could be further developed. Most of the extensions proposed below are currently being investigated at SICS.

- The maintenance planning and scheduling approaches presented in Papers D–F could be further developed in that risks in the form of statistical distributions of failure probabilities could be explicitly introduced, and the cost of corrective maintenance could be balanced against the cost of preventive maintenance.
- The maintenance schedule optimization as presented in Papers E and F is for a single machine only. In practice, A production plant is composed of several machines, where maintenance activities as well as the effects of breakdown are correlated. One possible extension of the work presented in Papers E and F would be to consider several machines, connected in e.g. a series-parallel fashion. Also, there are obvious advantages in performing maintenance for several machines at the same time, since this maintenance can be done in parallel, minimizing the total down time of the plant. This type of maintenance has been considered previously in, for example, [144, 172, 216, 284, 286].
- Paper D focuses on demonstrating that it is possible to reduce the number of maintenance stops when stochastic maintenance in the form of individual subsystem accumulators are present. The planning methodology could benefit from more advanced models and solution approaches for both the railway circulation problem and for the maintenance planning subproblem. More rigorous approaches at solving the former problem can be found in e.g. [162]. The latter problem is discussed in great detail in Paper E–F. Although the focus is on a different deployment area, the railway domain could equally benefit from the proposed solution for maintenance planning.
- Finally, for a company maintaining several units, there are side constraints which are present in the real world but not considered in this thesis. For example, the availability of spare parts and labor, as well as the travel plans for repair crews, are in many cases crucial for timely

maintenance. For the railway domain, the resource constraints which are present in the workshop (a limited number of tracks, and the configuration of equipment mounted at a track) should also be considered.

5.6 Conclusions

The goal of the work included in the thesis was to solve real industrial, combinatorial, problems, and for the work to have a substantial practical impact. In this goal, we acknowledged the fact that optimization approaches for real combinatorial problem solving can fail due to unrealistic and inaccurate models, and a lack of understanding of the real industrial problem and the environment in which deployed software is going to be used. To avoid this, we have worked continuously in close collaboration with industrial partners to understand the application and its specific constraints. In solving the problems described in Papers A–F, we have used several different methods from computer science and artificial intelligence depending on the problem type and our estimates of the difficulty of solving the problems to optimality. The thesis work has also had substantial impact in industry and academia.

Bibliography

[1] L. ABENI, *Server Mechanisms for Multimedia Applications*, Tech. Report RETIS TR98-01, Scuola Superiore S. Anna, Pisa, Italy, 1998.

[2] L. ABENI AND G. BUTTAZZO, *Integrating Multimedia Applications in Hard Real-Time Systems*, in Proceedings of the 19th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1998, pp. 4–13.

[3] *AbsInt*. Web page, http://www.absint.com/stackanalyzer/.

[4] W. AFZAL, R. TORKAR, AND R. FELDT, *A systematic review of search-based testing for non-functional system properties*, Information and Software Technology, 51 (2009), pp. 957–976.

[5] J. ALANDER, T. MANTERE, G. MOGHADAMPOUR, AND J. MATILA, *Searching Protection Relay Response Time Extremes Using Genetic Algorithm — Software Quality by Optimization*, in Proceedings of the International Conference on Advances in Power System Control, Operation and Management, vol. 1, 1997, pp. 95–99.

[6] A. ALFIERI, R. GROOT, L. KROON, AND A. SCHRIJVER, *Efficient Circulation of Railway Rolling Stock*, Transportation Science, 40 (2006), pp. 378–391.

[7] H. ALLAOUI, S. LAMOURI, A. ARTIBA, AND E. AGHEZZAF, *Simultaneously scheduling n jobs and the preventive maintenance on the two-machine flow shop to minimize the makespan*, International Journal of Production Economics, 1 (2008), pp. 161–167.

[8] T. ALMGREN, N. ANDRÉASSON, D. ANEVSKI, M. PATRIKSSON, A.-B. STRÖMBERG, AND J. SVENSSON, *Optimization of opportunistic replacement activities: A case study in the aircraft industry*, tech. report, Chalmers University, Göteborg, May 2008.

[9] T. ALMGREN, N. ANDRÉASSON, M. PATRIKSSON, A.-B. STRÖMBERG, AND A. WOJCIECHOWSKI, *The replacement problem: A polyhedral and complexity analysis*, Tech. Report Preprint 2009:4, Chalmers University, Göteborg, January 2009.

[10] B. ALPERN, R. HOOVER, B. K. ROSEN, P. F. SWEENEY, AND F. K. ZADECK, *Incremental Evaluation of Computational Circuits*, in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California, January 1990, pp. 32–42.

[11] I. ALSYOUF, *Maintenance practices in swedish industries: Survey results*, International Journal of Production Economics, (2009), pp. 212–223.

[12] L. ANDEREGG, S. EIDENBENZ, M. GANTENBEIN, C. STAMM, D. S. TAYLOR, B. WEBER, AND P. WIDMAYER, *Train Routing Algorithms: Concepts, Design Choices, and Practical*, in Proceedings of the 5th Workshop on Algorithm Engineering and Experiments, Society for Industrial and Applied Mathematics, 2003, pp. 106–118.

[13] J. H. ANDERSON, S. RAMAMURTHY, AND K. JEFFAY, *Real-Time Computing with Lock-Free Shared Objects*, ACM Transactions on Computer Systems, 15 (1997), pp. 134–165.

[14] J. ANDERSSON, J. HUSELIUS, C. NORSTRÖM, AND A. WALL, *Extracting Simulation Models from Complex Embedded Real-Time Systems*, in Proceedings of the International Conference on Software Engineering Advances, ICSEA'06, IEEE, Oct. 2006.

[15] N. ANDRÉASSON, *Optimization of opportunistic replacement activities in deterministic and stochastic multi-component systems*, licentiate thesis, Chalmers University, May 2004.

[16] N. ANDRÉASSON, *Utvärdering av matematiska modeller för optimering av opportunistiska underhållsbeslut för flygmotorer*, tech. report, Chalmers University, Göteborg, May 2005.

[17] D. L. APPLEGATE, R. E. BIXBY, V. CHVATAL, AND W. J. COOK, *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, January 2007.

[18] K. R. APT, *Principles of Constraint Programming*, Cambridge University Press, August 2003.

[19] *Arcticus Systems*. Web page, http://www.arcticus-systems.com.

[20] N. AUDSLEY, A. BURNS, R. DAVIS, K. TINDELL, AND A. WELLINGS, *Fixed Priority Pre-emptive Scheduling: an Historical Perspective*, Real-Time Systems, 8 (1995), pp. 129–154.

[21] N. AUDSLEY, A. BURNS, M. RICHARDSON, K. TINDELL, AND A. J. WELLINGS, *Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*, Software Engineering Journal, 8 (1993), pp. 284–292.

[22] N. C. AUDSLEY, A. BURNS, M. F. RICHARDSON, AND A. J. WELLINGS, *Real-Time Scheduling: the Deadline-Monotonic Approach*, in Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software, 1991.

[23] K. R. BAKER AND D. TRIETSCH, *Principles of Sequencing and Scheduling*, John Wiley \& Sons, Inc., 2009.

[24] T. P. BAKER, *A Stack Based Resource Allocation Policy for Real-Time Processes*, in Proceedings of the 11th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1990.

[25] T. P. BAKER, *Stack-Based Scheduling of Realtime Processes*, Real-Time Systems, 3 (1991), pp. 67–99.

[26] R. BARLOW AND L. HUNTER, *Optimum Preventive Maintenance Policies*, Operations Research, 8 (1960), pp. 90–100.

[27] R. E. BARLOW AND F. PROSCHAN, *Mathematical Theory of Reliability*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[28] I. BATE, P. NIGHTINGALE, AND A. CERVIN, *Establishing timing requirements and control attributes for control loops in Real-Time systems*, in Proceedings of the 15th Euromicro Conference on Real-time Systems, IEEE Computer Society, 2003, pp. 121–128.

[29] G. BEHRMANN, A. DAVID, J. HÅKANSSON, M. HENDRIKS, K. G. LARSEN, P. PETTERSSON, AND W. YI, *UPPAAL 4.0*, in Proceedings of the International Conference on Quantitative Evaluation of Systems, 2006.

[30] N. BELDICEANU AND E. CONTEJEAN, *Introducing Global Constraints in CHIP*, Mathematical and Computer Modelling, 20 (1994), pp. 97–123.

[31] M. BENGTSSON, *On Condition Based Maintenance and its Implementation in Industrial Settings*, PhD thesis, Mälardalen University Press Dissertation, November 2007.

[32] M. BERG, *General trigger-off replacement procedures for two-unit systems*, Naval Research Logistics, (1978), pp. 15–29.

[33] M. BEVILACQUA AND M. BRAGLIA, *The Analytic Hierarchy Process Applied to Maintenance Strategy Selection*, Reliability Engineering & System Safety, 70 (2000), pp. 71–83.

[34] B. BLANCHARD, D. VERMA, AND E. PETERSON, *Maintainability: A Key to Effective Serviceability and Maintenance Management*, John Wiley & Sons, Inc., New York, USA, 1995.

[35] M. BOHLIN, Y. LU, J. KRAFT, P. KREUGER, AND T. NOLTE, *Best-Effort Simulation-Based Timing Analysis using Hill-Climbing with Random Restarts*, in Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society, August 2009.

[36] B. BOUZY AND T. CAZENAVE, *Computer Go: an AI Oriented Survey*, Artificial Intelligence, 132 (2001), pp. 39–103.

[37] M. P. BOYCE, *Gas Turbine Engineering Handbook*, Gulf Professional Publishing, Boston, MA, USA, 2006.

[38] S. A. BRANDT, S. BANACHOWSKI, C. LIN, AND T. BISSON, *Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes*, in Proceedings of the 24th IEEE Real-Time Systems Symposium, IEEE Computer Society, 2003.

[39] L. C. BRIAND, Y. LABICHE, AND M. SHOUSHA, *Stress Testing Real-Time Systems with Genetic Algorithms*, in Proceedings of the 2005 conference on Genetic and evolutionary computation, Washington DC, USA, 2005, ACM, pp. 1021–1028.

[40] L. C. BRIAND, Y. LABICHE, AND M. SHOUSHA, *Using genetic algorithms for early schedulability analysis and stress testing in real-time systems*, Genetic Programming and Evolvable Machines, 7 (2006), pp. 145–170.

[41] R. BRIS, *Parallel simulation algorithm for maintenance optimization based on directed acyclic graph*, Reliability Engineering & System Safety, 93 (2008), pp. 874–884.

[42] R. BRIS, E. CHÂTELET, AND F. YALAOUI, *New method to minimize the preventive maintenance cost of series-parallel systems*, Reliability Engineering & System Safety, 82 (2003), pp. 247–255.

[43] G. BUDAI-BALKE, R. DEKKER, AND R. NICOLAI, *A Review of Planning Models for Maintenance and Production*, tech. report, Erasmus University Rotterdam, Econometric Institute, 2006. Econometric Institute Report 2006-44.

[44] A. BURNS, K. TINDELL, AND A. WELLINGS, *Effective Analysis for Engineering Real-Time Fixed Priority Schedulers*, IEEE Transactions on Software Engineering, 21 (1995), pp. 475–480.

[45] A. BURNS AND A. WELLINGS, *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*, Addison Wesley Longmain, March 2001.

[46] G. C. BUTTAZZO, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[47] D. G. CATTRYSSE, M. SALOMON, AND L. N. VAN WASSENHOVE, *A Set Partitioning Heuristic for the Generalized Assignment Problem*, European Journal of Operations Research, 72 (1994), pp. 167–174.

[48] K. CHATTERJEE, D. MA, R. MAJUMDAR, T. ZHAO, T. HENZINGER, AND J. PALSBERG, *Stack size analysis for interrupt-driven programs*, in Proceedings of the 10th Annual International Static Analysis Symposium, June 2003.

[49] B. CHEN, C. N. POTTS, AND G. J. WOEGINGER, *A review of machine scheduling: Complexity, algorithms and approximability*, vol. 3 of Handbook of Combinatorial Optimization, Kluwer Academic Publishers, 1998.

[50] D. I. CHO AND M. PARLAR, *A survey of maintenance models for multi-unit systems*, European Journal of Operations Research, 51 (1991), pp. 1–23.

[51] H. CHO, B. RAVINDRAN, AND E. D. JENSEN, *A Space-Optimal Wait-Free Real-Time Synchronization Protocol*, in Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, July 2005.

[52] G. CLARKE AND J. W. WRIGHT, *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points*, Operations Research, 12 (1964), pp. 568–581.

[53] J. COETZEE, *Maintenance*, Trafford Publishing, 2004.

[54] S. A. COOK, *The Complexity of Theorem-Proving Procedures*, in Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio, 1971, pp. 151–158.

[55] R. COOKE AND J. PAULSEN, *Concepts for Measuring Maintenance Performance and Methods for Analysing Competing Failure Modes*, Reliability Engineering & System Safety, 5 (1997), pp. 135–141.

[56] J. CORDEAU, P. TOTH, AND D. VIGO, *A Survey of Optimization Models for Train Routing and Scheduling*, Transportation Science, 32 (1998), pp. 380–404.

[57] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, MIT Press, Cambridge, MA, USA, second ed., 2001.

[58] D. G. CORNEIL, *Graph-Theoretic Concepts in Computer Science*, Springer Verlag, 2005, ch. Lexicographic Breadth First Search - A Survey, pp. 1–19.

[59] M. COUTINHO, J. RUFINO, AND C. ALMEIDA, *Response Time Analysis of Asynchronous Periodic and Sporadic Tasks Sheduled by a Fixed Priority Preemptive Algorithm*, in Proceedings of the 20th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2008, pp. 156–167.

[60] G. B. DANTZIG AND J. H. RAMSER, *The Truck Dispatching Problem*, Management Science, 6 (1959), pp. 80–91.

[61] *Dash Optimization*. Web page, http://www.dashoptimization.com.

[62] A. DAVENPORT AND E. TSANG, *Solving Constraint Satisfaction Sequencing Problems by Iterative Repair*, in The 1st International Conference on The Practical Application of Constraint Technologies and Logic Programming, april 1999, pp. 345–357.

[63] R. DAVIS, N. MERRIAM, AND N. TRACEY, *How Embedded Applications using an RTOS can stay within On-chip Memory Limits*, in Proceedings of the Work-in-Progress and Industrial Experience Session, Euromicro Conference on Real-Time Systems, June 2000.

[64] R. I. DAVIS, A. BURNS, R. J. BRIL, AND J. J. LUKKIEN, *Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised*, Real-Time Systems, 35 (2007), pp. 239–272.

[65] J. M. V. DE CARVALHO, *LP Models for Bin Packing and Cutting Stock Problems*, European Journal of Operations Research, 141 (2002), pp. 253–273.

[66] K. S. DE SMIDT-DESTOMBES, M. C. VAN DER HEIJDEN, AND A. VAN HARTEN, *Availability of k-out-of-N systems under block replacement sharing limited spares and repair capacity*, International Journal of Production Economics, 107 (2007), pp. 404–421.

[67] D. DECOTIGNY AND I. PUAUT, *ARTISST: An Extensible and Modular Simulation Tool for Real-Time Systems*, in Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.

[68] R. DEKKER, *Integrating optimisation, priority setting, planning and combining of maintenance activities*, European Journal of Operations Research, 82 (1995), pp. 225–240.

[69] R. DEKKER, *Applications of Maintenance Optimization Models: A Review and Analysis*, Reliability Engineering & System Safety, 51 (1996), pp. 229–240.

[70] R. DEKKER, R. V. EGMOND, J. FRENK, AND R. WILDEMAN, *A General Approach for the Coordination of Maintenance Frequencies*, Econometric Institute Report EI 9539-/A, Erasmus University Rotterdam, Econometric Institute, 1995.

[71] R. DEKKER AND P. SCARF, *On the Impact of Optimisation Models in Maintenance Decision Making: the State of the Art*, Reliability Engineering & System Safety, 60 (1998), pp. 111–119.

[72] R. DEKKER AND E. SMEITINK, *Preventive Maintenance at Opportunities of Restricted Duration*, Tech. Report 1991-38, Faculteit der Economische Wetenschappen en Econometrie, Vrije Universiteit, Amsterdam, April 1991.

[73] R. DEKKER, R. E. WILDEMAN, AND F. A. VAN DER DUYN SCHOUTEN, *A Review of Multi-Component Maintenance Models with Economic Dependence*, Mathematical Methods of Operations Research, 45 (1997), pp. 411–435.

[74] R. DEKKER, R. E. WILDEMAN, AND R. VAN EGMOND, *Joint Replacement in an Operational Planning Phase*, European Journal of Operations Research, 91 (1996), pp. 74–88.

[75] B. DICKMAN, S. EPSTEIN, AND Y. WILAMOWSKY, *A mixed integer linear programming formulation for multi-component deterministic opportunistic replacement*, The Journal of the Operational Research Society of India, (1991), pp. 165–175.

[76] E. W. DIJKSTRA, *A Note on Two Problems in Connexion With Graphs*, Numerische Mathematik, 1 (1959), pp. 269–271.

[77] R. DOBRIN AND G. FOHLER, *Reducing the Number of Preemptions in Fixed Priority Scheduling*, in Proceedings of the 16th Euromicro Conference on Real-time Systems, IEEE Computer Society, 2004.

[78] S. EDGAR AND A. BURNS, *Statistical Analysis of WCET for Scheduling*, in Proceedings of the 22th IEEE Real-Time Systems Symposium, IEEE Computer Society, 2001.

[79] M. EHRGOTT, *Multicriteria Optimization*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[80] C. EKELIN, *An Optimization Framework for Scheduling of Embedded Real-Time Systems*, PhD thesis, Chalmers University of Technology, 2004.

[81] J. ENGBLOM, *Static Properties of Commercial Embedded Real-Time Programs, and Their Implication for Worst-Case Execution Time Analysis*, in Proceedings of the 5th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, June 1999.

[82] J. ENGBLOM AND A. ERMEDAHL, *Modeling Complex Flows for Worst-Case Execution Time Analysis*, in Proceedings of the 21st IEEE Real-Time Systems Symposium, IEEE Computer Society, 2000, pp. 163–174.

[83] J. ENGBLOM, A. ERMEDAHL, M. NOLIN, J. GUSTAFSSON, AND H. HANSSON, *Worst-Case Execution-Time Analysis for Embedded Real-Time Systems*, International Journal on Software Tools for Technology Transfer, 4 (2003), pp. 437–455.

[84] T. ERLEBACH, M. GANTENBEIN, D. HÜRLIMANN, G. NEYER, A. PAGOURTZIS, P. PENNA, K. SCHLUDE, K. STEINHÖFEL, D. S. TAYLOR, AND P. WIDMAYER, *On the Complexity of Train Assignment Problems*, in Proceedings of the 12th International Symposium on Algorithms and Computation, London, UK, 2001, Springer-Verlag, pp. 390–402.

[85] *Evidence Srl*. Web page, http://www.evidence.eu.com.

[86] P. FIOOLE, L. KROON, G. MARÓTI, AND A. SCHRIJVER, *A rolling stock circulation model for combining and splitting of passenger trains*, European Journal of Operations Research, 174 (2006), pp. 1281–1297.

[87] *FlexRay*. Web page, http://www.flexray.com.

[88] M. S. FOX, *Constraint-directed Search: A Case Study of Job-Shop Scheduling*, Morgan Kaufmann Publishers, 1987.

[89] M. L. FREDMAN AND R. E. TARJAN, *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms*, Journal of the ACM, 34 (1987), pp. 596–615.

[90] D. R. FULKERSON AND O. A. GROSS, *Incidence Matrices and Interval Graphs*, Pacific Journal of Mathematics, 15 (1965), pp. 835–855.

[91] D. FURCY AND S. KOENIG, *Limited Discrepancy Beam Search*, in Proceedings of the 19th International Joint Conference on Artificial Intelligence, L. P. Kaelbling and A. Saffiotti, eds., Professional Book Center, July 2005, pp. 125–131.

[92] H. A. GABBAR, H. YAMASHITA, K. SUZUKI, AND Y. SHIMADA, *Computer-aided RCM-based plant maintenance management system*, Robotics and Computer-Integrated Manufacturing, 19 (2003), pp. 449–458.

[93] P. GAI, G. LIPARI, AND M. D. NATALE, *Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip*, in Proceedings of the 22th IEEE Real-Time Systems Symposium, IEEE Computer Society, 2001.

[94] G. GALANTE AND G. PASSANNANTI, *An exact algorithm for preventive maintenance planning of series-parallel systems*, Reliability Engineering & System Safety, 94 (2009), pp. 1517–1525.

[95] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, January 1979.

[96] A. GARG AND S. DESHMUKH, *Maintenance Management: Literature Review and Directions*, Journal of Quality in Maintenance Engineering, 12 (2006), pp. 205–238.

[97] I. P. GENT AND T. WALSH, *Towards an understanding of hill-climbing procedures for SAT*, in AAAI, 1993, pp. 28–33.

[98] I. P. GENT AND T. WALSH, *The search for satisfaction*, tech. report, Dept. of Computer Science, University of Strathclyde, 1999.

[99] I. GERTSBAKH, *Reliability Theory With Applications to Preventive Maintenance*, Springer-Verlag, Berlin, Germany, 2005.

[100] R. GHATTAS AND A. DEAN, *Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis*, in Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2007.

[101] P. C. GILMORE AND R. E. GOMORY, *A Linear Programming Approach to the Cutting-Stock Problem*, Operations Research, 9 (1961), pp. 849–859.

[102] F. GLOVER AND M. LAGUNA, *Tabu Search*, in Modern Heuristic Techniques for Combinatorial Optimization, C. R. Reeves, ed., McGraw-Hill, 1995, ch. 3, pp. 70–150.

[103] D. E. GOLDBERG, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Professional, January 1989.

[104] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, no. 57 in Annals of Discrete Mathematics, Elsevier, February 2004.

[105] J. B. GOODENOUGH AND L. SHA, *The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks*, in Proceedings of the 2nd International Workshop on Real-time Ada Issues, New York, NY, USA, 1988, ACM, pp. 20–31.

[106] S. K. GOYAL AND M. I. KUSY, *Determining Economic Maintenance Frequency for a Family of Machines*, The Journal of the Operational Research Society, 36 (1985), pp. 1125–1128.

[107] J. GU, P. W. PURDOM, J. FRANCO, AND B. W. WAH, *Algorithms for the Satisfiability (SAT) Problem: a Survey*, in Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem, vol. 35, 1997, pp. 19–152.

[108] J. C. P. GUTIERREZ AND M. G. HARBOUR, *Schedulability Analysis for Tasks with Static and Dynamic Offsets*, in Proceedings of the 19th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1998.

[109] J. C. P. GUTIERREZ AND M. G. HARBOUR, *Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities*, in Proceedings

of the 24th IEEE Real-Time Systems Symposium, IEEE Computer Society, 2003.

[110] J. GUTIÈRREZ, J. GARCÍA, AND M. HARBOUR, *Best-Case Analysis for Improving the Worst-Case Schedulability Test for Distributed Hard Real-Time Systems*, in Proceedings of the 10th Euromicro Conference on Real-time Systems, Los Alamitos, CA, USA, 1998, IEEE Computer Society, pp. 35–44.

[111] Y. HANI, L. AMODEO, F. YALAOUI, AND H. CHEN, *Simulation based optimization of a train maintenance facility*, Journal of Intelligent Manufacturing, 19 (2008), pp. 293–300.

[112] Y. HANI, H. CHEHADE, L. AMODEO, AND F. YALAOUI, *Simulation based optimization of a train maintenance facility model using genetic algorithms*, in Service Systems and Service Management, 2006 International Conference on, vol. 1, 2006, pp. 513–518.

[113] H. HANSSON, M. NOLIN, AND T. NOLTE, *Beating the Automotive Code Complexity Challenge*, in National Workshop on High-Confidence Automotive Cyber-Physical Systems, April 2008.

[114] P. E. HART, N. J. NILSSON, AND B. RAPHAEL, *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*, SIGART Bulletin, (1972), pp. 28–29.

[115] W. D. HARVEY AND M. L. GINSBERG, *Limited Discrepancy Search*, in Proceedings of the 14th International Joint Conference on Artificial Intelligence, 1995, pp. 607–615.

[116] R. HECKMANN AND C. FERDINAND, *Verifying Safety-Critical Timing and Memory-Usage Properties of Embedded Software by Abstract Interpretation*, in Proceedings of the Design, Automation and Test in Europe, March 2005.

[117] K. HÄNNINEN, J. MÄKI-TURJA, S. SANDBERG, J. LUNDBÄCK, M. LINDBERG, M. NOLIN, AND K.-L. LUNDBÄCK, *Framework for Real-Time Analysis in Rubus-ICE*, in Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation, IEEE, 2008.

[118] J. HUSELIUS, J. KRAFT, H. HANSSON, AND S. PUNNEKKAT, *Evaluating the Quality of Models Extracted from Embedded Real-Time Software*, in ECBS, IEEE Computer Society, 2007, pp. 577–585.

[119] A. HØYLAND AND M. RAUSAND, *System Reliability Theory - Models and Statistical Methods*, John Wiley & Sons, Inc., USA, 1994.

[120] *IEEE Standard 610.10-1994. Glossary of Computer Hardware Terminology*, IEEE, 1994. ISBN 1-55937-492-6.

[121] *ILOG CPLEX*. Web page, http://www.ilog.com, 2009.

[122] *ISO 11898-1:2003. Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, International Organization for Standards, 2003.

[123] V. JAYABALAN AND D. CHAUDHURI, *Cost Optimization of Maintenance Scheduling for a System with Assured Reliability*, IEEE Transactions on Reliability, 41 (1992), pp. 21–25.

[124] J. JHANG AND S. SHEU, *Optimal age and block replacement policies for a multi-component system with failure interaction*, Journal of Systems Science, 5 (2000), pp. 593–603.

[125] M. JOSEPH AND P. PANDYA, *Finding Response Times in a Real-Time System*, The Computer Journal, 29 (1986), pp. 390–395.

[126] Y. JUANG, S. LIN, AND H. KAO, *A knowledge management system for series-parallel availability optimization and design*, Expert Systems with Applications, 34 (2008), pp. 181–193.

[127] A. B. KAHN, *Topological Sorting of Large Networks*, Communications of the ACM, 5 (1962), pp. 558–562.

[128] N. KARMARKAR, *A New Polynomial-Time Algorithm for Linear Programming*, Combinatorica, (1984), pp. 373–395.

[129] W. KAROUI, M.-J. HUGUET, P. LOPEZ, AND W. NAANAA, *YIELDS: A Yet Improved Limited Discrepancy Search for CSPs*, in CPAIOR, P. V. Hentenryck and L. A. Wolsey, eds., vol. 4510 of Lecture Notes in Computer Science, Springer, 2007, pp. 99–111.

[130] R. M. KARP, *Reducibility Among Combinatorial Problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, 1972, pp. 85–103.

[131] D. I. KATCHER, H. ARAKAWA, AND J. K. STROSNIDER, *Engineering and Analysis of Fixed Priority Schedulers*, IEEE Transactions on Software Engineering, 19 (1993), pp. 920–934.

[132] H. KELLERER, U. PFERSCHY, AND D. PISINGER, *Knapsack Problems*, Springer, Berlin, Germany, 2004.

[133] L. G. KHACHIAN, *A Polynomial Algorithm in Linear Programming*, Doklady Akademii Nauk SSSR, (1979), pp. 1093–1096. English translation in Doklady Mathematics 20, 191-194, 1979.

[134] U. KHAN AND I. BATE, *WCET Analysis of Modern Processors Using Multi-Criteria Optimisation*, in Search Based Software Engineering, International Symposium on, vol. 0, Los Alamitos, CA, USA, 2009, IEEE Computer Society, pp. 103–112.

[135] K. KIM, J. L. DIAZ, L. L. BELLO, J. M. LOPEZ, C.-G. LEE, AND S. L. MIN, *An Exact Stochastic Analysis of Priority-Driven Periodic Real-Time Systems and Its Approximations*, IEEE Transactions on Computers, 54 (2005), pp. 1460–1466.

[136] S. KIRKPATRICK, C. D. GELATT, AND M. P. VECCHI, *Optimization by Simulated Annealing*, Science, 220 (1983), pp. 671–680.

[137] H. KOPETZ AND G. GRÜNSTEIDL, *TTP — A Protocol for Fault-Tolerant Real-Time Systems*, Computer, 27 (1994), pp. 14–23.

[138] R. E. KORF, *Improved Limited Discrepancy Search*, in Proceedings of the 8th Innovative Applications of Artificial Intelligence Conference, Menlo Park, CA, USA, 1996, AAAI Press, pp. 286–291.

[139] J. KRAFT, *RTSSim – A Simulation Framework for Complex Embedded Systems*, Tech. Report., Mälardalen University, March 2009.

[140] J. KRAFT, J. HUSELIUS, A. WALL, AND C. NORSTRÖM, *Extracting Simulation Models from Complex Embedded Real-Time Systems*, in Real-Time in Sweden 2007, August 2007.

[141] J. KRAFT, Y. LU, C. NORSTRÖM, AND A. WALL, *A Metaheuristic Approach for Best Effort Timing Analysis Targeting Complex Legacy Real-Time Systems*, in Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2008.

[142] P. KREUGER AND M. ARONSSON, *A Constraint Model for a Cyclic Time Personnel Routing and Scheduling Problem*, Technical Report T2001:04, SICS, September 2001.

[143] P. KREUGER, M. CARLSSON, T. SJÖLAND, AND E. ÅSTRÖM, *Sequence Dependent Task Extensions for Trip Scheduling*, Technical Report T2001:04, SICS, May 2001.

[144] R. LAGGOUNE, A. CHATEAUNEUF, AND D. AISSANI, *Opportunistic Policy for Optimal Preventive Maintenance of a Multi-Component System in Continuous Operating Units*, Computers & Chemical Engineering, In Press, Corrected Proof (2009).

[145] A. H. LAND AND A. G. DOIG, *An Automatic Method of Solving Discrete Programming Problems*, Econometrica, 28 (1960), pp. 497–520.

[146] C. LEE, M. WANG, G. CHASLOT, J. HOOCK, A. RIMMEL, O. TEYTAUD, S. TSAI, S. HSU, AND T. HONG, *The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments*, IEEE Transactions on Computational Intelligence and AI in Games, (2009).

[147] C. G. LEE, K. LEE, J. HAHN, Y. M. SEO, S. L. MIN, R. HA, S. HONG, C. Y. PARK, M. LEE, AND C. S. KIM, *Bounding Cache-Related Preemption Delay for Real-Time Systems*, IEEE Transactions on Software Engineering, 27 (2001), pp. 805–826.

[148] J. P. LEHOCZKY, *Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines*, in Proceedings of the 11th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1990, pp. 201–213.

[149] J. LEOHOLD, *Communication Requirements for Automotive Systems*. Keynote presentation at 5th IEEE International Workshop on Factory Communication Systems, September 2004. Available 2009-08-17 at http://www.ict.tuwien.ac.at/wfcs2004/downloads/keynote_leohold.pdf.

[150] R. LEWIS, *Metaheuristics can Solve Sudoku Puzzles*, Journal of Heuristics, 13 (2007), pp. 387–401.

[151] O. LHOMME, *Consistency Techniques for Numeric CSPs*, in Proceedings of the 13th International Joint Conference on Artificial Intelligence, 1993, pp. 232–238.

[152] C. LIU AND J. LAYLAND, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, 20 (1973), pp. 46–61.

[153] *Live Devices ETAS Group*. Web page, http://en.etasgroup.com/products/rta/.

[154] A. K. MACKWORTH, *Consistency in networks of relations*, Artificial Intelligence, 8 (1977), pp. 99–118.

[155] B. MAGGARD AND D. RHYNE, *Total Productive Maintenance: A Timely Integration of Production and Maintenance*, Production & Inventory Management Journal, 33 (1992), pp. 6–10.

[156] J. MÄKI-TURJA AND M. NOLIN, *Efficient Implementation of Tight Response-Times for Tasks with Offsets*, Real-Time Systems, 40 (2008), pp. 77–116.

[157] M. MARCHESOTTI, M. MIGLIARDI, AND R. PODESTA, *A measurement-based analysis of the responsiveness of the Linux kernel*, in 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 2006, pp. 10 pp.–408.

[158] K. MARRIOTT AND P. J. STUCKEY, *Programming with Constraints, An Introduction*, MIT Press, 1998.

[159] M. MARSEGUERRA, E. ZIO, AND L. PODOFILLINI, *Condition-Based Maintenance Optimization by Means of Genetic Algorithms and Monte Carlo Simulation*, Reliability Engineering & System Safety, 77 (2002), pp. 151–165.

[160] S. MARTELLO AND P. TOTH, *Knapsack Problems*, John Wiley & Sons Ltd., Chichester, England, 1990.

[161] G. MARÓTI AND L. KROON, *Maintenance Routing for Train Units: The Transition Model*, Transportation Science, 39 (2005), pp. 518–525.

[162] G. MARÓTI AND L. KROON, *Maintenance routing for train units: The interchange model*, Computers & Operations Research, 34 (2007), pp. 1121–1140.

[163] J. D. MATTINGLY, *Elements of Gas Turbine Propulsion*, McGraw-Hill, Inc, Singapore, third ed., 1996.

[164] J. J. MCCALL, *Maintenance Policies for Stochastically Failing Equipment: A Survey*, Management Science, 11 (1965), pp. 493–524.

[165] T. A. MCKEE AND F. MCMORRIS, *Topics in Intersection Graph Theory*, no. QA 166.105.M34 in SIAM Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, 1999.

[166] K. MCKONE AND E. WEISS, *TPM: Planned and Autonomous Maintenance: Bridging the Gap between Practice and Research*, Production and Operations Management, 7 (1998), pp. 335–351.

[167] *Micro Digital Inc.: smx® Special Features*. Available 2009-10-20 at: http://www.smxinfo.com/rtos/kernel/smxfeatr.pdf.

[168] B. MIDDHA, M. SIMPSON, AND R. BARUA, *MTSS: Multi Task Stack Sharing for Embedded Systems*, in Proceedings of the of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, San Francisco, CA, Sept 2005.

[169] J. MOUBRAY, *Reliability-Centered Maintenance - RCM II*, Industrial Press, Inc., New York, second ed., 1997.

[170] D. MURTHY, A. ATRENS, AND J. ECCLESTON, *Strategic maintenance management*, Journal of Quality in Maintenance Engineering, 8 (2002), pp. 287 – 305.

[171] *NAVAIR 00-25-403 Management Manual: Guidelines for the naval aviation reliability-centered maintenance process*, July 2005. Direction of Commander, Naval Air Systems Command.

[172] A. NEELAKANTESWARA RAO AND B. BHADURY, *Opportunistic maintenance of multi-equipment system: a case study*, Quality and Reliability Engineering International, 16 (2000), pp. 487–500.

[173] G. L. NEMHAUSER AND L. A. WOLSEY, *Integer and Combinatorial Optimization*, Wiley-Interscience, November 1999.

[174] R. NICOLAI AND R. DEKKER, *Optimal Maintenance of Multi-Component Systems: a Review*, tech. report, Erasmus University Rotterdam, Econometric Institute, Aug. 2006. Econometric Institute Report 2006-26.

[175] N. J. NILSSON, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill Pub. Co., 1971.

[176] T. NOLTE, H. HANSSON, M. NOLIN, AND S. PUNNEKKAT, *Timing Analysis of CAN-Based Automotive Communication Systems*, CRC Press, Taylor & Francis Group, 2008.

[177] T. NOLTE, H. HANSSON, AND C. NORSTRÖM, *Effects of Varying Phasings of Message Queuings in CAN Based Systems*, in Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications, IEEE Computer Society, 2002, pp. 261–266.

[178] T. NOLTE, H. HANSSON, AND C. NORSTRÖM, *Minimizing CAN Response-Time Jitter by Message Manipulation*, in Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, September 2002, pp. 197–206.

[179] T. NOLTE, M. NOLIN, AND H. HANSSON, *Real-Time Server-Based Communication for CAN*, IEEE Transactions on Industrial Informatics, 1 (2005), pp. 192–201.

[180] T. NOLTE, G. RODRÍGUEZ-NAVAS, J. PROENZA, S. PUNNEKKAT, AND H. HANSSON, *Towards Analyzing the Fault-Tolerant Operation of Server-CAN*, in Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation, IEEE, September 2005.

[181] F. S. NOWLAN AND H. F. HEAP, *Reliability-Centered Maintenance*, U.S. Department of Commerce, Springfield, V.A., 1978.

[182] R. OVERMAN, *RCM, Condition Monitoring, or Both?*, Maintenance Technology, (2002), pp. 25–28.

[183] R. PAGE, *Maintenance Management and Delay Reduction*, Maintenance & Asset Management, 17 (2002), pp. 5–13.

[184] J. C. PALENCIA AND M. G. HARBOUR, *Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems*, in Proceedings of the 20th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1999, pp. 328–339.

[185] J. PALSBERG AND D. MA, *A typed interrupt calculus*, in Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, London, UK, 2002, Springer-Verlag, pp. 291–310.

[186] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization; Algorithms and Complexity*, Dover Publications, 1998.

[187] C. M. PAPADIMITRIOU, *Computational Complexity*, Addison-Wesley, Reading, Massachusetts, 1994.

[188] C. L. PAPE, *Constraint Propagation in Planning and Scheduling*, tech. report, Robotics Laboratory, Department of Computer Science, Stanford University, Palo Alto, CA, 1991.

[189] R. PASCUAL, V. MERUANE, AND P. REY, *On the effect of downtime costs and budget constraint on preventive and replacement policies*, Reliability Engineering & System Safety, 93 (2008), pp. 144–151.

[190] A. E. PAULL, *Linear Programming, A Key to Optimum Newsprint Production*, Pulp and Paper Magazine of Canada, 57 (1956), pp. 85–90.

[191] M. PEETERS AND L. KROON, *Circulation of Railway Rolling Stock: A Branch-and-Price Approach*, Computers & Operations Research, 35 (2008), pp. 538–556.

[192] H. PHAM AND H. WANG, *Optimal $(s;T)$ opportunistic maintenance of a k-out-of-n : G system with imperfect PM and partial failure*, Naval Research Logistics, 3 (2000), pp. 223–239.

[193] W. P. PIERSKALLA AND J. A. VOELKER, *A survey of maintenance models: The control and surveillance of deteriorating systems*, Naval Research Logistics, 23 (1976), pp. 353–388.

[194] L. M. PINTELON AND L. F. GELDERS, *Maintenance management decision making*, European Journal of Operations Research, 58 (1992), pp. 301–317.

[195] P. P. PUSCHNER AND A. BURNS, *Guest Editorial: A Review of Worst-Case Execution-Time Analysis*, Real-Time Systems, 18 (2000), pp. 115–128.

[196] P. P. PUSCHNER AND C. KOZA, *Calculating the Maximum Execution Time of Real-Time Programs*, Real-Time Systems, 1 (1989), pp. 159–176.

[197] M. PĚNIČKA, A. K. STRUPCHANSKA, AND D. BJØRNER, *Train Maintenance Routing*, in Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems, L'Harmattan Hongrie, May 2003.

[198] N. RACHANIOTIS AND C. PAPPIS, *Preventive maintenance and upgrade system: Optimizing the whole performance system by components' replacement or rearrangement*, International Journal of Production Economics, 1 (2008), pp. 236–244.

[199] R. RACU AND R. ERNST, *Scheduling Anomaly Detection and Optimisation for Distributed Systems with Preemptive Task-Sets*, in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2006, pp. 325–334.

[200] R. RAJAMANI, J. WANG, AND K. Y. JOENG, *Condition Based Maintenance for Aircraft Engines*, in *Proceedings of the ASME Turbo Expo*, 2004. Paper no. GT2004-54127.

[201] K. RAMAMRITHAM, *Allocation and Scheduling of Complex Periodic Tasks*, in Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE Computer Society, May 1990, pp. 108–115.

[202] H. RAMAPRASAD AND F. MUELLER, *Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks*, in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2006.

[203] *Rapita Systems*. Web page, http://www.rapitasystems.com, 2008.

[204] O. REDELL, *Analysis of Tree-Shaped Transactions in Distributed Real-Time Systems*, in Proceedings of the 16th Euromicro Conference on Real-time Systems, IEEE Computer Society, 2004, pp. 239–248.

[205] O. REDELL AND M. SANFRIDSON, *Exact Best-Case Response Time Analysis of Fixed Priority Scheduled Tasks*, in Proceedings of the 14th Euromicro Conference on Real-time Systems, IEEE Computer Society, 2002, pp. 165–172.

[206] O. REDELL AND M. TÖRNGREN, *Calculating Exact Worst Case Response Times for Static Priority Scheduled Tasks with Offsets and Jitter*, in Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, Los Alamitos, CA, USA, September 2002, IEEE Computer Society, pp. 164–172.

[207] J. REGEHR, *Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults*, in Proceedings of the 23rd IEEE Real-Time Systems Symposium, IEEE Computer Society, 2002.

[208] J. REGEHR, A. REID, AND K. WEBB, *Eliminating Stack Overflow by Abstract Interpretation*, ACM Transactions in Embedded Computing Systems, 4 (2005), pp. 751–778.

[209] J.-C. RÉGIN, *A Filtering Algorithm for Constraints of Difference in CSPs*, in Proceedings of the 12th National Conference on Artificial Intelligence, Menlo Park, California, 1994, AAAI Press, pp. 362–367.

[210] J.-C. REGIN AND J.-F. PUGET, *A Filtering Algorithm for Global Sequencing Constraints*, in Principles and Practice of Constraint Programming, 1997, pp. 32–46.

[211] J. M. ROBSON, *The Complexity of Go*, in Information Processing 83, Proceedings of the 9th World Computer Congress, R. E. A. Mason, ed., North-Holland/IFIP, 1983, pp. 413–417.

[212] D. J. ROSE, R. E. TARJAN, AND G. S. LUEKER, *Algorithmic Aspects of Vertex Elimination on Graphs*, SIAM Journal of Computing, 5 (1976), pp. 266–283.

[213] G. T. ROSS AND R. M. SOLAND, *A Branch and Bound Algorithm for the Generalized Assignment Problem*, Mathematical Programming, 8 (1975), pp. 91–103.

[214] S. RUSSELL AND P. NORVIG, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Second ed., 2003.

[215] I. SABUNCUOGLU AND M. BAYIZ, *Job shop scheduling with beam search*, European Journal of Operations Research, 118 (1999), pp. 390–412.

[216] N. SADEH, *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling*, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1991.

[217] N. SADEH, S. OTSUKA, AND R. SCHNELBACH, *Predictive and reactive scheduling with the micro-boss production scheduling and control system*, in Proceedings of the IJCAI-93 Workshop on Knowledge-based Production Planning, Scheduling, & Control, Chambery, France, Aug 1993.

[218] *SAE JA1011: Evaluation Criteria for Reliability-Centered Maintenance (RCM) Processes*, August 1998. Society of Automotive Engineers.

[219] H. E. SAKKOUT AND M. WALLACE, *Probe backtrack search for minimal perturbation in dynamic scheduling*, Constraints, 5 (2000), pp. 359–388.

[220] M. SAKSENA AND Y. WANG, *Scalable Real-Time System Design Using Preemption Thresholds*, in Proceedings of the 21st IEEE Real-Time Systems Symposium, IEEE Computer Society, 2000.

[221] S. SAMII, S. RAFILIU, P. ELES, AND Z. PENG, *A Simulation Methodology for Worst-Case Response Time Estimation of Distributed Real-Time Systems*, in Proceedings of Design, Automation and Test in Europe, vol. 10-14, IEEE, Mar. 2008, pp. 556–561.

[222] K. SANDSTRÖM, C. ERIKSSON, AND G. FOHLER, *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*, in Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications Symposium, IEEE Computer Society, 1998, pp. 158–165.

[223] A. SANGIOVANNI-VINCENTELLI AND M. DI NATALE, *Embedded System Design for Automotive Applications*, Computer, 40 (2007), pp. 42–51.

[224] R. G. SARGENT, *Validation and verification of simulation models*, in Proceedings of the $36^{textrmth}$ conference on Winter simulation, Washington, D.C., 2004, Winter Simulation Conference, pp. 17–28.

[225] M. SAVELSBERGH, *A Branch-and-Price Algorithm for the Generalized Assignment Problem*, Operations Research, 45 (1997), pp. 831–841.

[226] P. A. SCARF, *On the Application of Mathematical Models in Maintenance*, European Journal of Operations Research, 99 (1997), pp. 493–506.

[227] A. SCHRIJVER, *Minimum Circulation of Railway Stock*, CWI Quarterly, 6 (1993), pp. 205–217.

[228] D. SCULLI AND A. W. SURAWEERA, *Tramcar Maintenance*, The Journal of the Operational Research Society, 30 (1979), pp. 809–814.

[229] B. SELMAN, H. A. KAUTZ, AND B. COHEN, *Noise strategies for local search*, in Proceedings of the 12th National Conference on Artificial Intelligence, Menlo Park, California, 1994, AAAI Press, pp. 337–343.

[230] L. SHA, T. ABDELZAHER, K.-E. ÅRZÉN, A. CERVIN, T. BAKER, A. BURNS, G. BUTTAZZO, M. CACCAMO, J. LEHOCZKY, AND A. K. MOK, *Real Time Scheduling Theory: A Historical Perspective*, Real-Time Systems, 28 (2004), pp. 101–155.

[231] L. SHA, J. P. LEHOCZKY, AND R. RAJKUMAR, *Priority Inheritance Protocols: An Approach To Real-Time Synchronisation*, Tech. Report CMU-CS-87-181, Computer Science Department, Carnegie-Mellon University, 1987.

[232] L. SHA, R. RAJKUMAR, AND J. LEHOCZKY, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Transactions on Computers, 39 (1990), pp. 1175–1185.

[233] M. SJÖDIN AND H. HANSSON, *Improved Response-Time Analysis Calculations*, in Proceedings of the 19th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1998.

[234] S. SMITH, *OPIS: A Methodology and Architecture for Reactive Scheduling*, in Intelligent Scheduling, M. Zweben and M. Fox, eds., Morgan Kaufmann, 1994.

[235] S. SMITH, N. KENG, AND K. KEMPF, *Exploiting Local Flexibility During Execution of Pre-computed Schedules*, Tech. Report CMU-RI-TR-90-13, Robotics Institute, Pittsburgh, PA, June 1990.

[236] S. SMITH, *Reactive Scheduling Systems*, in Intelligent Scheduling Systems, D. Brown and W. Scherer, eds., Kluwer Press, 1995.

[237] N. SORTRAKUL, H. NACHTMANN, AND C. CASSADY, *Genetic algorithms for integrated preventive maintenance planning and production scheduling for a single machine*, Computers in Industry, 56 (2005), pp. 161–168.

[238] R. SOSIC AND J. GU, *Efficient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem*, IEEE Transactions on Knowledge and Data Engineering, 6 (1994), pp. 661–668.

[239] C. SRISKANDARAJAH, A. K. S. JARDINE, AND C. K. CHAN, *Maintenance Scheduling of Rolling Stock Using a Genetic Algorithm*, The Journal of the Operational Research Society, 49 (1998), pp. 1130–1145.

[240] J. A. STANKOVIC, *Misconceptions About Real-Time Computing*, Computer, 21 (1988), pp. 10–19.

[241] J. A. STANKOVIC AND K. RAMAMRITHAM, eds., *Tutorial: hard real-time systems*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[242] J. A. STANKOVIC, K. RAMAMRITHAM, AND M. SPURI, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[243] J. STASCHULAT, S. SCHLIECKER, AND R. ERNST, *Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay*, in Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2005.

[244] A.-B. STRÖMBERG AND T. ALMGREN, *Optimering av underhållsplaner leder till strategier för utvecklingsprojekt*, in Logistikutveckling - teori möter praktik, artiklar från PLANs Forsknings- och tillämpningskonferens, August 2008, pp. 169–180.

[245] J. S. TAN AND M. A. KRAMER, *A General Framework for Preventive Maintenance Optimization in Chemical Process Operations*, Computers & Chemical Engineering, 21 (1997), pp. 1451–1469.

[246] R. TARJAN, *Depth-first search and linear graph algorithms*, SIAM Journal of Computing, 1 (1972), pp. 146–160.

[247] R. E. TARJAN, *Edge-Disjoint Spanning Trees and Depth-First Search*, Acta Informatica, 6 (1976), pp. 171–185.

[248] *Tidorum*. Web page, http://www.tidorum.fi/bound-t/.

[249] K. TINDELL, *Using Offset Information to Analyse Static Priority Preemptively Scheduled Task Sets*, Tech. Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[250] K. TINDELL AND A. BURNS, *Fixed Priority Scheduling of Hard Real-time Multi-media Disk Traffic*, The Computer Journal, 37 (1994), pp. 691–697.

[251] K. TINDELL AND J. CLARK, *Holistic Schedulability Analysis for Distributed Hard Real-Time Systems*, Microprocessing and Microprogramming, 40 (1994), pp. 117–134.

[252] K. TINDELL, H. HANSSON, AND A. WELLINGS, *Analysing Real-Time Communications: Controller Area Network (CAN)*, in Proceedings of the 16th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1995.

[253] K. W. TINDELL, A. BURNS, AND A. J. WELLINGS, *An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks*, Real-Time Systems, 6 (1994), pp. 133–151.

[254] M. TLILI, S. WAPPLER, AND H. STHAMER, *Improving Evolutionary Real-Time Testing*, in Proceedings of the 8th annual conference on Genetic and evolutionary computation, Seattle, Washington, USA, 2006, ACM, pp. 1917–1924.

[255] P. TOTH AND D. VIGO, *The Vehicle Routing Problem*, SIAM Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, 2001.

[256] A. TSANG, *Strategic dimensions of maintenance management*, Journal of Quality in Maintenance Engineering, 8 (12 April 2002), pp. 7–39.

[257] E. TSANG, *Foundations of Constraint Satisfaction*, Academic Press, 1994.

[258] R. UEHARA AND Y. UNO, *Efficient Algorithms for the Longest Path Problem*, in Algorithms and Computation, 2005, pp. 871–883.

[259] *Unicoi Systems*. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.

[260] *UPPAAL Website*. Web page, http://www.uppaal.com, 2008.

[261] J. S. USHER, A. H. KAMAL, AND W. H. SYED, *Cost optimal preventive maintenance and replacement scheduling*, IIE Transactions, 30 (1998), pp. 1121–1128.

[262] C. VALDEZ-FLORES AND R. M. FELDMAN, *A survey of preventive maintenance models for stochastically deteriorating single-unit systems*, Naval Research Logistics, 36 (1989), pp. 419–446.

[263] G. VAN DIJKHUIZEN AND A. VAN HARTEN, *Optimal clustering of frequency-constrained maintenance jobs with shared set-ups*, European Journal of Operations Research, 99 (1997), pp. 552–564.

[264] S. VITTAL, P. HAJELA, AND A. JOSHI, *Review of Approaches to Gas Turbine Life Management*, in *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, vol. 2, 2004, pp. 876–886.

[265] J. P. WALSER, *Integer Optimization by Local Search*, vol. 1637 of Lecture Notes in Artificial Intelligence, Springer, 1999.

[266] T. WALSH, *Depth-bounded Discrepancy Search*, in Proceedings of the 15th International Joint Conference on Artificial Intelligence, August 1997, pp. 1388–1393.

[267] H. WANG, *A Survey of Maintenance Policies of Deteriorating Systems*, European Journal of Operations Research, 139 (2002), pp. 469–489.

[268] Y. WANG AND M. SAKSENA, *Scheduling Fixed-Priority Tasks with Preemption Threshold*, in Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications Symposium, IEEE Computer Society, 1999.

[269] J. WATKINS, *Across the Board: The Mathematics of Chessboard Problems*, Princeton University Press, 2004.

[270] J. WEGENER AND M. GROCHTMANN, *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing*, Real-Time Systems, 15 (1998), pp. 275–298.

[271] J. WEGENER AND F. MUELLER, *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints*, Real-Time Systems, 21 (2001), pp. 241–268.

[272] D. B. WEST, *Introduction to Graph Theory*, Prentice Hall, first ed., 1996.

[273] R. WILDEMAN, R. DEKKER, AND A. SMIT, *Combining Maintenance Activities in an Operational Planning Phase: a Dynamic Programming Approach*, "Papers" Series 9424-a, Erasmus University of Rotterdam - Econometric Institute, 1994.

[274] R. E. WILDEMAN AND R. DEKKER, *Dynamic influences in multi-component maintenance*, Quality and Reliability Engineering International, 13 (1997), pp. 199–207.

[275] R. E. WILDEMAN, R. DEKKER, AND A. C. J. M. SMIT, *A Dynamic Policy for Grouping Maintenance Activities*, European Journal of Operations Research, 99 (1997), pp. 530–551.

[276] R. WILHELM, J. ENGBLOM, A. ERMEDAHL, N. HOLSTI, S. THESING, D. WHALLEY, G. BERNAT, C. FERDINAND, R. HECKMANN, T. MITRA, F. MUELLER, I. PUAUT, P. PUSCHNER, J. STASCHULAT, AND P. STENSTRÖM, *The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools*, ACM Transactions in Embedded Computing Systems, 7 (2008), pp. 1–53.

[277] W. L. WINSTON, *Introduction to Mathematical Programming: Applications and Algorithms*, Duxbury Resource Center, 2003.

[278] T. WIREMAN, *World Class Maintenance Management*, Industrial Press, Inc., 1990.

[279] M. WÄRJA, P. SLOTTNER, AND M. BOHLIN, *Customer Adapted Maintenance Plan (CAMP), a Process for Optimization of Gas Turbine Maintenance*, in *Proceedings of the ASME Turbo Expo*, 2008. Paper no. GT2008-50240.

[280] J. XU AND D. L. PARNAS, *Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations*, IEEE Transactions on Software Engineering, 3 (1990), pp. 360–369.

[281] Z. YAMAYEE, K. SIDENBLAD, AND M. YOSHIMURA, *A Computationally Efficient Optimal Maintenance Scheduling Method*, IEEE Transactions on Power Apparatus and Systems, PAS-102 (1983), pp. 330–338.

[282] T. YATO AND T. SETA, *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles*, IEICE Transactions on Fundamentals of Electronics, Communication and Computer Sciences, 86 (2003), pp. 1052–1060.

[283] W. ZHANG, *State-space search*, Springer, 1999.

[284] X. ZHENG AND N. FARD, *A Maintenance Policy for Repairable Systems Based on Opportunistic Failure-Rate Tolerance*, IEEE Transactions on Reliability, 40 (1991), pp. 237–244.

[285] X. ZHENG AND N. FARD, *Hazard-Rate Tolerance Method for an Opportunistic-Replacement Policy*, IEEE Transactions on Reliability, 1 (1992), pp. 13–20.

[286] X. ZHOU, L. XI, AND J. LEE, *A dynamic opportunistic maintenance policy for continuously monitored systems*, Journal of Quality in Maintenance Engineering, 12 (2006), pp. 294–305.

[287] X. ZHOU, L. XI, AND J. LEE, *Reliability-centered predictive maintenance scheduling for a continuously monitored system subject to degradation*, Reliability Engineering & System Safety, 92 (2007), pp. 530–534.

[288] X. ZHOU, L. XI, AND J. LEE, *Opportunistic preventive maintenance scheduling for a multi-unit series system based on dynamic programming*, International Journal of Production Economics, 118 (2009), pp. 361–366.

II

Included Papers

Chapter 6

Paper A: Determining maximum stack usage in preemptive shared stack systems

Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson and Mikael Nolin.

In Proceedings of the 27th IEEE Real-Time Systems Symposium. December 5–8, 2006, Rio de Janeiro, Brazil.

Abstract

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based on dynamic (run-time) properties.

We also show how to safely approximate the exact stack usage by using static (compile time) information about the system model and the underlying run-time system on a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system.

Comprehensive evaluations show that our technique significantly reduces the amount of stack memory needed compared to existing analysis techniques. For typical task sets a decrease in the order of 70% is typical.

6.1 Introduction

In conventional multitasking systems, each thread of execution (task) has its own allocated execution stack. In systems with a large number of tasks a large number of stacks are required. Hence the total amount of RAM needed for the stacks can grow exceedingly large.

Stack sharing is a memory model in which several tasks share one common run-time stack. It has been shown that stack sharing can result in memory savings [9, 16] compared to the conventional stack model. The shared stack model is applicable to both non-preemptive as well as preemptive systems, and it is especially suitable in resource constrained embedded real-time systems with limited amount of memory. Stack sharing is currently supported by many commercial real-time kernels, e.g. [3, 18, 20, 32].

The traditional method to calculate the memory requirements for a shared run-time stack in preemptive systems is to sum the maximum stack usage of tasks in each preemption level and possibly consider additional overheads such as memory used by interrupts and context switches. A major drawback with the traditional calculation method is that it often results in over allocation of stack memory by presuming that all tasks with maximum stack usage in each priority level can preempt each other in a nested fashion during run-time. However, there may, in many cases, be no actual possibility for these tasks to preempt each other (e.g. due to explicit or implicit separation in time). Moreover, the possible preemptions may not be able to occur in a nested fashion.

Taking advantage of the fact that many real-time system exhibit a predictable temporal behavior, it is possible to identify feasible preemption scenarios, i.e., which preemptions can in fact occur, and whether they can occur in a nested fashion or not. Therefore, a more accurate stack analysis can be made. One example of a system that lends itself to such analysis is a hybrid, statically and dynamically, scheduled system. Such a system consists of an off-line scheduler producing the static schedule and a fixed priority scheduler (FPS) that dispatches tasks at run-time. The commercial operating system, Rubus OS by Arcticus Systems AB [3], supports such a system model. The Rubus OS is mainly used in resource-constrained embedded real-time systems. For instance, in the vehicular industry, Volvo Construction Equipment (VCE) [33], BAE Systems Hägglunds [15], and Haldex Traction Systems [13] all use the Rubus OS in their vehicles or components.

In this paper, we present the general problem of analyzing a shared system stack for resource constrained preemptive real-time systems. We provide a general and exact problem formulation applicable for preemptive systems

based on dynamic run-time properties. We also present an approximate stack analysis method to derive a safe upper bound on stack usage in static offset based, fixed priority and preemptive systems that use a shared stack. We evaluate and show that the proposed method gives significantly lower upper bounds on stack memory requirements than existing stack dimensioning methods for fixed priority systems.

Paper outline. Section 6.2 describes related work and sets the context for the contributions of this paper. In sections 6.3, 6.4, and 6.5 we present the exact formulation of determining the maximum stack usage and our safe approximation of the stack usage for our target system model. Section 6.6 presents an evaluation of our approximative analysis method, and Section 6.7 concludes the paper.

6.2 Related work

The notion of shared stack has been used in several publications to describe the ability to utilize either a common run-time stack or a pool of run-time stacks. For example, in [20], stack sharing is performed by having a pool of available stack areas. When a task starts executing, it fetches a stack from the pool, and returns it at termination. In [21], Middha *et al.* address stack sharing in the sense that the stack of a task can grow into the stack area of another task.

In this paper, we use the notion of stack sharing when several tasks use one common, statically allocated, run-time stack. This type of stack sharing can be efficiently implemented in systems where tasks have run-to-completion semantics, and do not suspend themselves. This type of stack sharing is supported by several commercial real-time operating systems, e.g. [3, 18, 32].

6.2.1 Stack analysis

In [4], Baker presents the Stack Resource Policy (SRP) that permits stack sharing among processes in static and in some dynamic priority preemptive systems. The basic method to determine the maximum amount of stack usage in SRP is to identify the maximum stack usage for tasks at each priority level (or preemption level) and then to sum up these maximums for each priority level. A safe upper bound (SPL) on the total stack usage using information about priority levels can formally be expressed as:

$$SPL = \sum_{l \in \textit{prio-levels}} \max_{i \in \textit{tasks with prio } l} (S_i) \tag{6.1}$$

where S_i is the maximum stack usage of task i.

Gai *et al.* [11] present SRP with preemption thresholds (SRPT). They present a procedure to minimize shared stack usage, without jeopardizing schedulability, by use of non-preemption groups for tasks using SRPT. They extend the work of Saksena and Wang [27] by taking the stack usage of tasks into account when establishing non-preemption groups.

In [9] Davis *et al.* address stack memory requirements by using non-preemption groups to reduce the amount of memory needed for a shared stack. They show that the number of preemption levels required for typical systems can be relatively small, while maintaining schedulability.

Although non-preemption groups can reduce the amount of RAM needed for a shared stack, the use of non-preemption groups affects a system by restricting the occurrences of preemptions, which can have a negative affect on schedulability. Also, the method we present in this paper can further reduce the system stack by performing our analysis after preemption groups have been assigned.

6.2.2 Preemption analysis

A large number of publications address preemption analysis for different reasons, see, e.g. [2, 7, 10, 17, 24, 25, 29]. For example, in [17] Lee *et al.* present a technique to bound cache-related preemption delays in fixed-priority preemptive systems. They account for task phasing and nested preemption patterns among tasks to establish an upper bound on the cache timing delay introduced by preemptions. Our work relates to theirs in the sense that we also investigate occurrences of nested preemption patterns. However, our objectives differ in that Lee *et al.* are mainly interested in timing delays caused by cache reloading and preemption patterns whereas we address shared memory requirements as an effect of nested preemption patterns.

In [10], Dobrin and Fohler present a method to reduce the number of preemptions in fixed priority based systems. They define three fundamental conditions that have to be satisfied in order for a preemption to occur. The same conditions form the basis of our upper bound method described in Section 6.5.

6.3 Stack analysis of preemptive systems

The primary purpose of an execution stack is to store local data which consists of variables and state registers, parameters to subroutines and return addresses. Real-time systems typically have a separate stack, statically allocated, for each

task. However, under certain conditions, tasks can share stack to achieve a lower overall memory footprint of the system.

In this paper we consider systems where a subset of tasks use a common, statically allocated, run-time stack. For this to be possible, we assume that a task only uses the stack between the start time of an instance and the finishing time of that instance, i.e., no data remains on the stack from one instance of a task to the next. Furthermore, we require non-interleaving task execution [4, 9]. If υ_j begins executing between the start and finish of υ_i, then υ_i is not allowed to resume execution until υ_j has finished. In practice, this is ensured by not allowing tasks to suspend themselves voluntarily, or to be suspended by blocking once they have started their execution. In practice this means that OS-primitives like `sleep()` and `wait_for_event()` cannot be used, and that any blocking on shared resources must be handled before execution start, e.g., with a semaphore protocol like immediate inheritance protocol [6].

We formally define the start and finishing time of a task instance υ_i, as follows:

st_i The absolute time when υ_i actually begins executing.

ft_i The absolute time when υ_i terminates its execution.

At any given point in time, the worst case total stack usage of the system equals the sum of the stack usage for each individual task instance. Thus, with $s_i(t)$ denoting the actual stack usage of υ_i at time t, the maximum stack usage of the system can be expressed as follows:

$$\max_{t \in \textit{time instant}} \sum_{\upsilon_i \in \textit{task instances}} s_i(t) \tag{6.2}$$

This corresponds to the amount of memory that must be statically allocated for the shared stack to ensure the absence of stack overflow errors. For some systems, e.g., non-preemptive, statically scheduled systems with simple task code, it might be possible to directly compute or estimate $s_i(t)$. In general, however, they are not directly computable before the system is executed.

We note that the total stack usage depends on three basic properties:

(i) the stack memory usage of each task instance

(ii) the possible preemptions that may occur between any two instances

(iii) the ways in which preemptions can be nested

Determining the stack memory usage of a single task instance requires knowledge of the possible control-flow paths within the task code [14]. In [5]

Brylow *et al.* present a static checker for interrupt driven software. The checker is able to calculate the stack size of assembler programs by producing a control-flow graph annotated with information about time, space, safety and liveness.

However, due to the difficulties in determining the exact stack usage at every point in time for a given task instance, shared-stack analysis methods often assume that whenever a task is preempted, it is preempted when it uses its maximum stack depth. We make the same assumption, and use S_i to denote the maximum stack usage for task instance υ_i. Thus, when υ_i and υ_j are instances of the same task, we have $S_i = S_j$. Bounds on maximum stack usage for a given task can be derived by abstract interpretation using tools such as AbsInt [1] and Bound-T [30].

In order to calculate the maximum stack usage of the full system, we need to account for all possible preemption patterns. Under the assumption of non-interleaving task execution, a task instance, υ_i, is preempted by another task instance, υ_j, if (and only if) the following holds:

$$st_i < st_j < ft_i \tag{6.3}$$

In particular, we are interested in chains of nested preemptions. We define a *preemption chain* to be a set $\{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ of task instances such that

$$st_1 < st_2 < \cdots < st_k < ft_k < ft_{k-1} < \cdots < ft_1 \tag{6.4}$$

Under the assumption that the worst case stack usage of a task occur when the task is preempted, the worst case stack usage SWC for a shared stack preemptive system can be expressed as follows:

$$SWC = \max_{PC \in \textit{preemption chains}} \sum_{\upsilon_i \in PC} S_i \tag{6.5}$$

This formulation, however, cannot be directly used for analyzing and dimensioning the shared system stack since it is based on the dynamic (only available at run-time) properties st_i and ft_i. To be able to statically analyze the system, one has to relate the static (compile-time) properties to these dynamic properties. This is done by establishing how the system model, scheduling policy, and run-time mechanism constrain the values of the actual start and finishing times. The problem can be viewed as a scheduling problem with the objective of maximizing the total stack usage of the schedule, subject to system constraints on how tasks are ordered in the schedule.

6.4 System model for hybrid scheduled systems

The system model we adopt is based on the commercial operating system Rubus OS by Arcticus Systems AB [3], which supports the execution of both time triggered and event triggered tasks. The Rubus OS is mainly intended for, and used in dependable resource-constrained embedded real-time systems.

The system model is a hybrid, static and dynamic, scheduled system where a subset of the tasks are dispatched by a static cyclic scheduler (time triggered tasks). The rest of the tasks are dispatched by events in the system (event triggered tasks). The static schedule is constructed off-line and a fixed priority scheduler (FPS) dispatches tasks at run-time. The event-triggered tasks can be categorized in two different classes: (i) event-triggered interrupts which have higher priority than the time-triggered tasks, and (ii) background scheduled event-triggered tasks which have lower priority than the time-triggered tasks.

The time triggered tasks share a common system stack. It is the objective of this paper to analyze, and ultimately dimension this shared system stack efficiently. The time-triggered subsystem is used to host safety critical applications. Hence, to isolate it from any erroneous event-triggered tasks, it uses its own stack.

6.4.1 Formal system model

The system model used in this paper can be seen as an offset based model with static offsets [12, 22, 23, 31], defined as follows: The system, Γ, consists of a set of k transactions $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by a periodic sequence of events with period T_i. For non-periodic, events T_i denotes the minimum inter-arrival time between two consecutive events. The activating events are mutually independent, i.e., phasing between them is arbitrary. A transaction, Γ_i, contains $|\Gamma_i|$ tasks, and each task may not be activated (released for execution) until a time, offset, elapses after the arrival of the activating event.

We use τ_{ij} to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the number of the task within the transaction. A task, τ_{ij}, is defined by a worst case execution time (C_{ij}), an offset (O_{ij}), a deadline (D_{ij}), maximum jitter (J_{ij}), maximum blocking from lower priority tasks (B_{ij}), and a priority (P_{ij}). Furthermore, S_{ij} is used to denote the maximum stack usage of τ_{ij}. The system model is formally

expressed as:

$$\begin{aligned}\Gamma :=& \{\langle \Gamma_1, T_1\rangle, \ldots, \langle \Gamma_k, T_k\rangle\}\\ \Gamma_i :=& \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\}\\ \tau_{ij} :=& \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij}, S_{ij}\rangle\end{aligned}$$

We assume that the system is schedulable and that the worst case response-time for each task, (R_{ij}), has been calculated [23]. Due to the non-interleaving criterion for stack sharing, we require that tasks exhibit run-to-completion semantics when activated, i.e., they cannot suspend themselves. An invocation of a task can be viewed as a function call from the operating system, and the invocation terminates when the function call returns. When tasks share the same priority, they are served on a first-come first-served basis.

We assume that if access to shared resources is not handled by the static scheduler by time separation, a resource sharing protocol where blocking is done before start of execution is employed (such as the stack resource protocol [4] or the immediate inheritance protocol [6]).

Relating back to Rubus OS, one can view the system as a transaction based system with one transaction, Γ_t, corresponding to the static schedule (time-triggered tasks) and any number of transactions corresponding to higher priority event triggered tasks (interrupts). For the even-triggered transactions there are no restrictions placed on offset, deadline or jitter, i.e., they can each be either smaller or greater than the period. Since Γ_t represents the static schedule, which is cyclical with period T_t, offset, jitter and deadline are less than the period, i.e., $O_{tj}, D_{tj}, J_{tj} \leq T_t$ for the time-triggered transaction. How a scheduler can generate a feasible schedule with interfering interrupts is described in [22, 28].

It is the objective of this paper to find a tight upper bound on the shared system stack for the tasks in the time-triggered transaction Γ_t. Task j belonging to Γ_t we will denote τ_{tj}. The tasks in the transaction can be preempted by other tasks in the transaction and by higher priority event triggered tasks.

6.5 Stack analysis of hybrid scheduled systems

In this section, we describe a polynomial time method to establish a safe upper bound on the shared stack usage for the system model described in Section 6.4. The upper bound is safe in the sense that the run-time stack can never exceed the calculated upper bound.

A safe upper-bound estimate of the exact problem can be found by using

offsets and maximum response times as approximations of actual start and finishing times. Generalizing the preemption criteria identified by Dobrin and Fohler [10], we form the binary relation $\tau_{ti} \prec \tau_{tj}$ with the interpretation that τ_{ti} may be preempted by τ_{tj}. The relation holds whenever (1) τ_{ti} can become ready before τ_{tj}, (2) τ_{ti} possibly finishes (i.e., has a response time) after the start of τ_{tj}, and (3) τ_{ti} has lower priority than τ_{tj}. The relation can now formally be defined as:

$$\tau_{ti} \prec \tau_{tj} \equiv O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti} \wedge P_{ti} < P_{tj} \tag{6.6}$$

Lemma 1. *The $\prec$ relation is a safe approximation of the possible preemptions between tasks in Γ_t. That is, if τ_{ti} can under any run-time circumstance be preempted by τ_{tj}, then $\tau_{ti} \prec \tau_{tj}$ will hold.*

Proof of Lemma 1. *Suppose that τ_{ti} is preempted by τ_{tj}. We show that this implies (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, (2) $O_{tj} < R_{ti}$, and (3) $P_{ti} < P_{tj}$.*

(3) follows directly from the preemption. Now let t be the time instant when τ_{tj} has finished blocking, which implies $t \leq O_{tj} + J_{tj} + B_{tj}$. Then a possibly empty interval $[t, st_{tj}]$ of execution with higher priority than τ_{tj} follows, in which τ_{ti} cannot execute because $P_{ti} < P_{tj}$. Since τ_{ti} must start before τ_{tj}, we can conclude that $st_{ti} < t$, which together with $O_{ti} \leq st_{ti}$ and $t \leq O_{tj} + J_{tj} + B_{tj}$ gives us $O_{ti} < O_{tj} + J_{tj} + B_{tj}$ and (1). From Equation 6.3 we have $st_{tj} < ft_{ti}$ and this together with $O_{tj} \leq st_{tj}$ and $ft_{ti} \leq R_{ti}$ leads to $O_{tj} < R_{ti}$ and (2), which completes the proof. □

The upper-bound problem can now be informally stated as finding the maximum stack usage of all possible preemption chains in Γ_t. This equals finding the time instant in the schedule which has a maximum stack usage, given the approximation of actual start and finishing times with offsets and response times respectively, and assuming that at all preemptions the preempted task uses its maximal stack.

A sequence Q of tasks is a *possible preemption chain* (PPC) if it holds that $\tau_{ti} \prec \tau_{tj}$ for all τ_{ti}, τ_{tj} in Q where τ_{ti} occurs before τ_{tj} in the sequence. The stack usage SU_Q of a PPC Q is the sum of the stack usage of the individual tasks in the chain, i.e., $SU_Q = \sum_{\tau_{ti} \in Q} S_{ti}$.

A straightforward computation of a safe upper bound for a set of tasks involves computing the stack usage for all PPCs. However, for a set of n tasks there exist $2^n - 1$ different PPCs in the worst case, which yields an exponential time complexity for an algorithm based on this idea. A more efficient algorithm can be constructed by first finding sets of tasks which all overlap in time

without regarding priorities. These sets can then be investigated, in turn, to find a PPC with maximal stack usage. We let the relation $\tau_{ti} \preceq \tau_{tj}$ hold whenever the semiclosed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$ intersect, or more formally:

$$\tau_{ti} \preceq \tau_{tj} \equiv O_{ti} < R_{tj} \wedge O_{tj} < R_{ti} \tag{6.7}$$

The relation $\preceq$ is a relaxation of the $\prec$ relation. That is, $\tau_{ti} \prec \tau_{tj} \rightarrow \tau_{ti} \preceq \tau_{tj}$. To see this, suppose that $\tau_{ti} \prec \tau_{tj}$ which implies $O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti}$, according to Equation 6.6. Since $O_{tj} + J_{tj} + B_{tj} \leq R_{tj}$ follows from the notion of response time, we have $O_{ti} < R_{tj} \wedge O_{tj} < R_{ti}$, which also is the definition of $\tau_{ti} \preceq \tau_{tj}$.

We can now define an *overlap set* K_r as a set of tasks where:

$$\forall \tau_{ti}, \tau_{tj} \in K_r : \tau_{ti} \preceq \tau_{tj}$$

The stack usage SU_{K_r} of an overlap set K_r is defined as the maximum stack usage SU_Q of all PPCs Q where $Q \subseteq K_r$:

$$SU_{K_r} = \max_{\forall Q \subseteq K_r : PPC(Q)} (SU_Q) \tag{6.8}$$

K_r is *maximal*, if and only if, there exists no overlap set, K_s, such that $K_r \subset K_s$.

Lemma 2. *For any PPC Q, there exists a maximal overlap set K_r such that $Q \subseteq K_r$.*

Proof of Lemma 2. *From the definitions of a PPC and the $\prec$ and $\preceq$ relations, we know that for all tasks $\tau_{ti} \prec \tau_{tj}$ in Q it also holds that $\tau_{ti} \preceq \tau_{tj}$, and thus Q is an overlap set. Then, either Q is maximal, or it can become maximal by extending it with additional tasks. In either case, the lemma holds.* □

In all, the algorithm for computing the upper bound PUB on the maximum stack usage for a set of tasks Γ_t can be summarized as follows:

1. Find the maximal overlap sets in Γ_t:
   $K = \{K_1, K_2, \ldots, K_k\}$.
2. For each of them, compute SU_{K_r} according to Equation 6.8.
3. The upper bound of the stack usage for Γ_t can now be computed as follows:
   $$SUB = \max_{\forall K_r \in K} (SU_{K_r}) \tag{6.9}$$

Informally, we start by finding all sets of tasks that can overlap in time based on their offsets and worst case response times, which safely approximates their actual start and finishing times. For each such set (K_i), we find all possible preemption chains (PPCs) by also taking task priorities and maximal jitter and blocking time into account, and compute the stack usage for each chain. The stack usage of K_i is the maximum stack usage of all its PPCs, and the maximum stack usage (SUB) of the system is then obtained by taking the maximum stack usage of every K_i.

6.5.1 Correctness

In order to claim correctness of our approximate stack analysis method, we have to show that it never underestimates the actual stack usage that can occur during run-time.

Theorem 1. *The value computed by the SUB algorithm is a safe upper bound on the actual worst case stack usage for tasks in Γ_t. Formally, $SWC \leq SUB$.*

Proof of Theorem 1. *Let $\Psi \subseteq \Gamma_t$ be the sequence of tasks instances participating in the preemption situation which cause the worst case stack usage, that is, $SWC = \sum_{\tau_{ti} \in \Psi} S_{ti}$. According to Lemma 1, we must have $\tau_{ti} \prec \tau_{tj}$ for tasks τ_{ti} and τ_{tj} that occur in that order in Ψ, and thus Ψ is a PPC with $SU_\Psi = SWC$. Then, Lemma 2 ensures that there exists a maximal overlap set K_r such that $\Psi \subseteq K_r$, and we have $SU_\Psi \leq SU_{K_r}$. Thus, $SWC \leq SU_{K_r} \leq SUB$, which concludes the proof.* □

6.5.2 Computational complexity

The relaxation of $\prec$ into interval intersection (Equation 6.7) allows us to efficiently compute an upper bound on the stack usage (Equation 6.9) by applying a polynomial longest path algorithm on the linearly-bounded number of maximal overlap sets.

To first see that the set of maximal overlap sets $K = \{K_1, K_2, \ldots, K_k\}$ contain at most n elements, i.e., $k \leq n$, consider the graph (Γ_t, E), where Γ_t is the set of vertices and $E = \{\tau_{ti}\tau_{tj} \mid (\tau_{ti} \preceq \tau_{tj}) \wedge \tau_{ti}, \tau_{tj} \in \Gamma_t\}$ is the set of edges. From Equation 6.7, we have that edges $\tau_{ti}\tau_{tj} \in E$ correspond to intersection of the semi-closed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$, and therefore the graph is an *interval graph* [19]. Because every interval graph is also *chordal* [19], all maximal complete subgraphs in (Γ_t, E), which correspond to all maximal overlap sets, can be found in linear time [26]. Furthermore, for

chordal graphs there exists at most n such sets, and thus we have at most n overlap sets [19].

The problem of finding the worst PPC within a single overlap set K_r is significantly easier than for an arbitrary set of tasks. Since it holds that $\tau_{ti} \preceq \tau_{tj}$ for all tasks $\tau_{ti}, \tau_{tj} \in K_r$, and therefore in particular that $O_{ti} < R_{tj}$ for all tasks in K_r, we need only look for a maximum stack usage chain Q where (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, and (2) $P_{ti} < P_{tj}$ for all tasks τ_{ti} and τ_{tj} in that order in Q to find the worst PPC. A directed graph consisting of tasks in K_r and arcs corresponding to properties (1) and (2) is acyclic, and for such graphs a longest-path type algorithm can be used to find the worst PPC [8]. There exist longest-path algorithms with a time complexity of $O(n + m)$, where n is the number of tasks and m is the number of possible preemptions, of which there are at most $n(n-1)/2$. Taking the maximum of a maximal PPC in each set, K_r, of which there are at most n, we will, therefore, find a maximum stack size PPC in at most $O(n^3)$ time.

6.6 Evaluation

We evaluate the efficiency of our proposed method to establish a safe upper bound on shared stack usage by randomly generating realistic sized task sets. The size, load and stack usage of the task sets are derived from a wheel-loader application by Volvo Construction Equipment [33]. We use three different methods to calculate the shared system stack usage:

SPL The traditional method to dimension a shared system stack by summing up the maximum stack usage in each priority level.

SUB The safe upper bound on the shared stack usage presented in Section 6.5

SLB A lower bound on on the shared stack usage, for each task set.

The lower bound is established using simple heuristics that tries to maximize shared stack usage by generating only feasible preemption scenarios for the task set, and thus, represents scenarios that definitely can occur. From all PPCs, the heuristic selects a sample set of roughly 500 chains. For each of them, the method tries to construct a feasible arrival pattern for the ET tasks and actual execution time values that cause an actual preemption between the tasks in the chain. The quality of this heuristic method degrades as the length of the chains or the total number of PPCs increases, which can be seen in the figures.

By establishing a safe upper bound and a feasible lower bound, we know that the actual worst case stack usage is bounded by SUB and SLB. The rea-

son for including SLB is to give an indication on the maximum amount of improvement there might be for SUB.

6.6.1 Simulation setup

In our simulator we generate random task sets as input to the stack analysis application. The task generator takes the following input parameters:

- Total number of TT (time triggered) tasks (default = 250)
- Total load of TT tasks (default = 60%)
- Minimum and maximum priorities of TT tasks (default = 1 and 32)
- Minimum and maximum stack usage of TT tasks (default = 128 and 2048)
- Total number of ET (event triggered) tasks (default = 8)
- Total load of ET tasks (default = 20%)
- The shortest possible minimum inter-arrival time of an ET task (default = 1,000)

The generated schedule for TT tasks is always 10,000 time units. All ET tasks have higher priority than TT tasks. The default values for the input parameters represent a base configuration derived from a real application [33].

Using these parameters a task set with the following characteristics is generated:

- Each TT offset (O_{ti}) is randomly and uniformly distributed between 0 and 10,000.
- Worst case execution times for TT tasks, C_{ti}, are initially randomly assigned between 1 and 1000 time units. The execution times get adjusted by multiplying all C_{ti} by a fraction, so that the the TT load (as defined by the input parameter) is obtained.
- TT priorities are assigned randomly between minimum and maximum value with a uniform distribution.

6.6.2 Results

Each diagram shows three graphs corresponding to the stack usage calculated by the three methods: SPL, SUB, and SLB. Each point in the graphs represents the mean value of 100 generated task sets. We also measured the 95% confidence interval for the mean values. These are not shown because of their small size (less than 7% of the y-value for each point). We also measured the CPU time to calculate an upper bound on shared stack usage for each generated task set. Using the method described in Section 6.5, the calculations took less than 63ms per task set, on an Intel Pentium 4, 2.8GHz with 512MB of RAM.

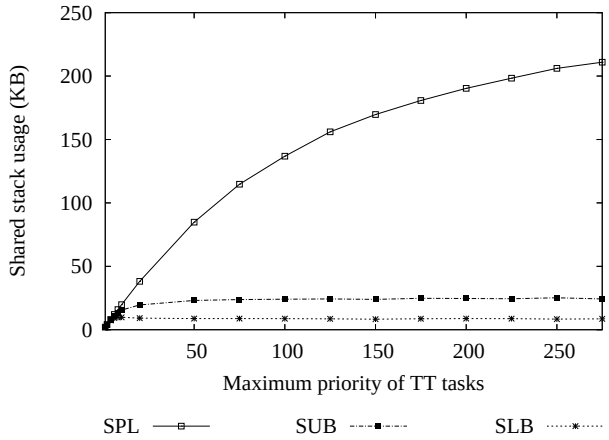


Figure 6.1: Varying the number of priority levels of TT tasks

In Fig. 6.1, we vary the maximum priority for TT tasks between 1 and 300, keeping the minimum priority at 1. This gives a distribution of possible priorities (priority levels), from 1 to n, where n is indicated by the x-axis. We see, in Fig. 6.2 which zooms in on Fig. 6.1, for maximum priorities up to 10, that the difference in stack usage between SPL and SUB is less noticeable with a low number of priority levels. However, for larger numbers of priority levels the difference is significant. SPL is not expected to flatten out before all tasks actually have unique priorities, whereas our method (SUB) flattens out significantly earlier. We conclude that the maximum number of tasks in any preemption chain is increasing very slowly (or not at all) when the number of TT tasks increases above a certain value, since the system load is constant.

In Fig. 6.3, we vary the maximum stack usage of each TT task between

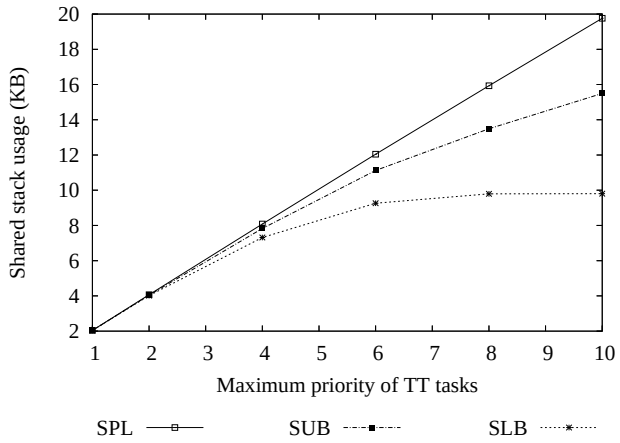


Figure 6.2: Varying the number of priority levels of TT tasks (zoom of Fig. 6.1)

128 bytes and 4096 bytes. We do this by assigning an initial stack of 128 bytes for each TT task, i.e. initially the stack size variation is zero. We then vary the stack size between 128 and 512 bytes, 128 and 1024 bytes, and so on. The diagram shows that SUB gives significantly lower values on shared stack usage than the traditional SPL. We also notice that an increase in stack variation scales up, linearly, the differences between SPL and SUB. The linearity is expected, since an increase in stack variation does not affect occurrences of possible preemptions in the system, i.e., possible nested preemptions are retained.

In Fig. 6.4 we vary the maximum number of TT tasks between 5 and 275. We see that the shared stack usage of the traditional SPL is dramatically increasing in the beginning. This is due to the fact that when the number of TT tasks is lower than the maximum priority of TT tasks (32), most TT tasks have unique priorities. SUB, on the other hand, increases much slower than SPL because the maximum number of tasks involved in any preemption chain is slowly increasing. SUB is expected to further approach SPL since increasing the number of tasks will increase the likelihood of larger number of tasks involved in the preemption chains.

In Fig. 6.5, we vary the total load of TT tasks between 10% (0.1) and 70% (0.7). The figure shows that the shared stack usage of SPL is constant, whereas, SUB is slowly increasing. SPL is expected to be constant since it is only af-

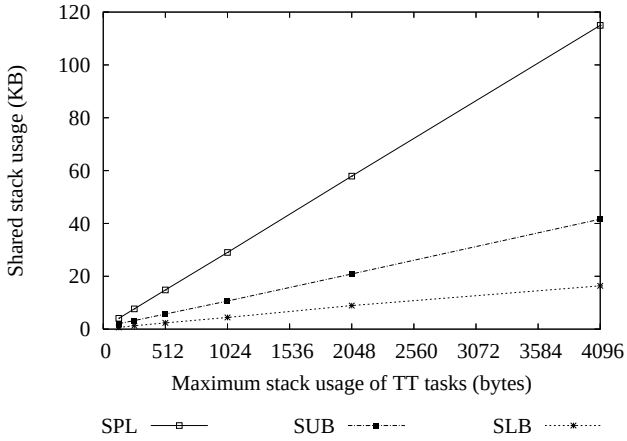


Figure 6.3: Varying stack usage of TT tasks

fected by the number of priority levels and unaffected by the actual preemptions that can occur in a system. The increase of SUB is due to increasing response-times of TT tasks when the TT load increases, which will increase the likelihood of larger number of tasks involved in nested preemptions.

6.7 Conclusions and future work

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based on dynamic (run-time) properties.

By approximating these run-time properties, together with information about the underlying run-time system, we present a method to safely approximate the maximum system stack usage at compile time. We do this for a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system. Such a system model provides lot of static information that we can use to estimate the dynamic start- and finishing-times. Our method finds the nested preemption pattern that results in the maximum shared stack usage. We prove that our method is a safe upper bound of the exact system stack usage and show that our method has a polynomial time complexity.

In a comprehensive simulation study, we evaluated our technique and com-

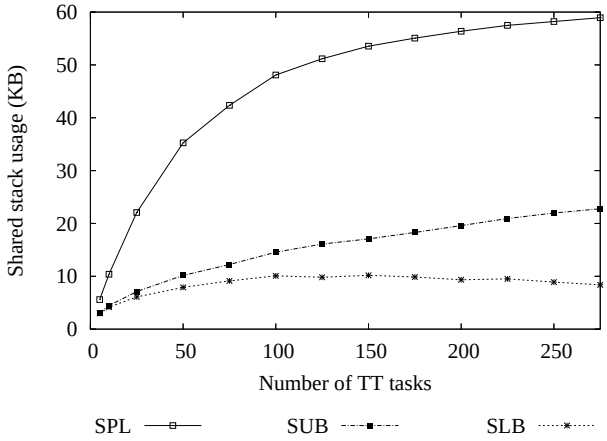


Figure 6.4: Varying the number of TT tasks

pared it to the traditional method to estimate stack usage. We find that our method significantly reduces the amount of stack memory needed. For realistically sized task sets, a decrease in the order of 70% is typical.

In this paper, we focused on a system model for a given commercial real-time operating system. In the future, we plan to extend our approximation method to a more general system model, to incorporate all the features of the general model for tasks with offsets [12]. Such an extension would make the presented analysis technique applicable to a wider range of systems.

Our current method could also be extended to account for other types of information that can further limit the number of possible preemptions. We currently only account for separation in time (offsets and response-times) between tasks. However, in many systems other types of information, such as precedence and mutual-exclusion relations may exists between tasks, thus limiting the possible preemptions.

The method presented here could also be used in synthesis and configuration tools that generate optimized systems from given application constraint. In this case, the results from our analysis can be used to guide optimization or heuristic techniques that try to map application functionality to run-time objects.

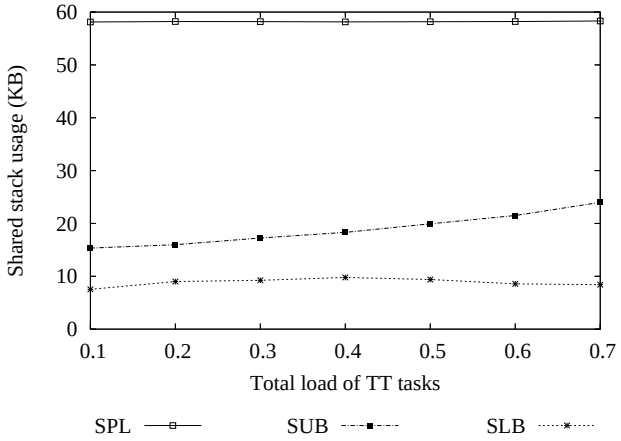


Figure 6.5: Varying the load of TT tasks

Bibliography

[1] *AbsInt*. Web page,
http://www.absint.com/stackanalyzer/.

[2] J. H. ANDERSON, S. RAMAMURTHY, AND K. JEFFAY, *Real-Time Computing with Lock-Free Shared Objects*, ACM Transactions on Computer Systems, 15 (1997), pp. 134–165.

[3] *Arcticus Systems*. Web page,
http://www.arcticus-systems.com.

[4] T. P. BAKER, *A Stack Based Resource Allocation Policy for Real-Time Processes*, in Proceedings of the 11th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1990.

[5] D. BRYLOW, N. DAMGAARD, AND J. PALSBERG, *Static checking of interrupt-driven software*, in Proceedings of the 23th International Conference on Software Engineering, May 2001.

[6] A. BURNS AND A. WELLINGS, *Real-Time Systems and Programming Languages*, Addison-Wesley, second ed., 1996, ch. 13.10.1 Immediate Ceiling Priority Inheritance.

[7] H. CHO, B. RAVINDRAN, AND E. D. JENSEN, *A Space-Optimal Wait-Free Real-Time Synchronization Protocol*, in Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, July 2005.

[8] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, MIT Press, Cambridge, MA, USA, second ed., 2001.

[9] R. DAVIS, N. MERRIAM, AND N. TRACEY, *How Embedded Applications using an RTOS can stay within On-chip Memory Limits*, in Proceedings of the Work-in-Progress and Industrial Experience Session, Euromicro Conference on Real-Time Systems, June 2000.

[10] R. DOBRIN AND G. FOHLER, *Reducing the Number of Preemptions in Fixed Priority Scheduling*, in Proceedings of the 16th Euromicro Conference on Real-time Systems, IEEE Computer Society, 2004.

[11] P. GAI, G. LIPARI, AND M. D. NATALE, *Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip*, in Proceedings of the 22th IEEE Real-Time Systems Symposium, IEEE Computer Society, 2001.

[12] J. C. P. GUTIERREZ AND M. G. HARBOUR, *Schedulability Analysis for Tasks with Static and Dynamic Offsets*, in Proceedings of the 19th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1998.

[13] *Haldex Traction Systems*. Web page, http://www.haldex-traction.com/.

[14] R. HECKMANN AND C. FERDINAND, *Verifying Safety-Critical Timing and Memory-Usage Properties of Embedded Software by Abstract Interpretation*, in Proceedings of the Design, Automation and Test in Europe, March 2005.

[15] *BAE Systems Hägglunds*. Web page, http://www.haggve.se.

[16] K. HÄNNINEN, J. LUNDBÄCK, K.-L. LUNDBÄCK, J. MÄKI-TURJA, AND M. NOLIN, *Efficient Event-Triggered Tasks in an RTOS*, in Proceedings of the 2005 International Conference on Embedded Systems and Applications, June 2005.

[17] C. G. LEE, K. LEE, J. HAHN, Y. M. SEO, S. L. MIN, R. HA, S. HONG, C. Y. PARK, M. LEE, AND C. S. KIM, *Bounding Cache-Related Preemption Delay for Real-Time Systems*, IEEE Transactions on Software Engineering, 27 (2001), pp. 805–826.

[18] *Live Devices ETAS Group*. Web page, http://en.etasgroup.com/products/rta/.

[19] T. A. MCKEE AND F. MCMORRIS, *Topics in Intersection Graph Theory*, no. QA 166.105.M34 in SIAM Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, 1999.

[20] *Micro Digital Inc.: smx® Special Features*. Available 2009-10-20 at: http://www.smxinfo.com/rtos/kernel/smxfeatr.pdf.

[21] B. MIDDHA, M. SIMPSON, AND R. BARUA, *MTSS: Multi Task Stack Sharing for Embedded Systems*, in Proceedings of the of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, San Francisco, CA, Sept 2005.

[22] J. MÄKI-TURJA, K. HÄNNINEN, AND M. NOLIN, *Efficient Development of Real-Time Systems Using Hybrid Scheduling*, in International Conference on Embedded Systems and Applications, June 2005.

[23] J. MÄKI-TURJA AND M. NOLIN, *Fast and tight response-times for tasks with offsets*.

[24] H. RAMAPRASAD AND F. MUELLER, *Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks*, in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2006.

[25] J. REGEHR, *Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults*, in Proceedings of the 23rd IEEE Real-Time Systems Symposium, IEEE Computer Society, 2002.

[26] D. J. ROSE AND R. E. TARJAN, *Algorithmic Aspects of Vertex Elimination*, in STOC '75: Proceedings of the 7th annual ACM symposium on Theory of computing, New York, NY, USA, 1975, ACM Press, pp. 245–254.

[27] M. SAKSENA AND Y. WANG, *Scalable Real-Time System Design Using Preemption Thresholds*, in Proceedings of the 21st IEEE Real-Time Systems Symposium, IEEE Computer Society, 2000.

[28] K. SANDSTRÖM, C. ERIKSSON, AND G. FOHLER, *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*, in Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications Symposium, IEEE Computer Society, 1998, pp. 158–165.

[29] J. STASCHULAT, S. SCHLIECKER, AND R. ERNST, *Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay*, in Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2005.

[30] *Tidorum*. Web page, http://www.tidorum.fi/bound-t/.

[31] K. TINDELL, *Using Offset Information to Analyse Static Priority Preemptively Scheduled Task Sets*, Tech. Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[32] *Unicoi Systems*. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.

[33] *Volvo Construction Equipment*. Web page, http://www.volvoce.com.

Chapter 7

Paper B: Bounding shared-stack usage in systems with offsets and precedences

Markus Bohlin, Kaj Hänninen, Jukka Mäki-Turja, Jan Carlson and Mikael Nolin.

In Proceedings of the 20th Euromicro Conference on Real-Time Systems.

July 2–4, 2008, Prague, Czech Republic.

Abstract

The paper presents two novel methods to bound the stack memory used in preemptive, shared stack, real-time systems. The first method is based on branch-and-bound search for possible preemption patterns, and the second one approximates the first in polynomial time. The work extends previous methods by considering a more general task-model, in which all tasks can share the same stack. In addition, the new methods account for precedence and offset relations. Thus, the methods give tight bounds for a large set of realistic systems. The methods have been implemented and a comprehensive evaluation, comparing our new methods against each other and against existing methods, is presented. The evaluation shows that our exact method can significantly reduce the amount of stack memory needed.

7.1 Introduction

In order to limit the amount of RAM set aside for stack-memory in embedded systems, many RTOSes provide means to execute multiple tasks on a single, shared, stack (e.g. Rubus [3], Fusion [28], Erika [10], SMX [17], etc.). In order to make maximum use of this ability to share stack-memory we need methods to properly dimension the memory allocated to the stack. This paper shows how to exploit commonly available knowledge of precedence and offsets between tasks to calculate a tight upper bound on the amount of stack-memory used.

In shared stack systems, one stack-frame is added to the system's stack for each level of preemption. Thus, the maximum stack-usage occurs during some worst-case preemption pattern. In simple task models (commonly used in real-time scheduling theory), where tasks are assumed to be independent, any preemption pattern is possible — thus we have to pessimistically assume that all tasks may be active and preempted at the point where they use the most stack. The system's maximum stack-usage thus becomes $\sum S_i$ (where S_i denotes the maximum stack-usage of task i). The consequence is that in these models the benefits of using a shared stack is limited.

In many systems we have information that let us deduce that some preemption patterns are impossible. For example, in a system where multiple tasks share the same priority, no preemptions among these tasks are possible (assuming FIFO scheduling within a priority level and an early-blocking resource allocation protocol such as the immediate inheritance protocol). In this case, the system's maximum stack-usage becomes $\sum_p \max_p(S_i)$ (where p denotes a priority level and $\max_p$ maximizes over the tasks within that priority level). If the number of priority levels is low enough, this type of analysis can provide a much lower bound on stack usage than the above sum over all tasks. Davis *et al.* describes this type of stack analysis and generalize it to allow non-preemption groups to be defined [8].

However, limiting the scheduler by lowering the number of priority levels or manually defining non-preemption groups has drawbacks, since it limits the schedulability of the system and places extra burden on the engineers to define non-preemption groups. Also, in many systems there is even more information available that would allow us to further reduce the possible preemptions in the system.

In this paper we present novel techniques to exploit information about precedence and offset relations between tasks to further limit the number of possible preemption-patterns. We perform a system wide preemption analysis

to find the worst case preemption pattern with respect to stack usage. This allows us to calculate a tight bound on the amount of stack memory needed in the system. The intuition behind the techniques is that tasks that have precedence relations will, under certain conditions, never preempt each other, and tasks with offset relations may only preempt each other if the response-time of the first task is longer than the offset to the second task. Thus, a prerequisite to perform our analysis is that the response-time and release jitter are known for all tasks. We build our analysis on the transactional task-model introduced by Tindell [27] which was formalized and extended by Gutiérrez and Harbour [13]. Given the safe approximations of response-times and jitter resulting from the schedulability analysis presented by, e.g., Mäki-Turja and Nolin [20], we here present two methods to bound the system stack usage. We present one algorithm that searches the whole search space of possible preemptions which has exponential complexity, and a safe approximation method with polynomial complexity. We provide an evaluation of the two methods, comparing them with each other and with the method of summation over priority levels described above.

The transactional task-model allows for modeling of large, complex and realistic real-time systems. Hence, the methods presented have a clear practical value. The methods can be used in a verification/validation phase of system development in order to formally verify that stack overflow will not occur during runtime. The approximation method (due to its better run-time complexity) could also be used in optimizing allocation, mapping, and configuration tools that automate the process of allocating tasks to nodes in distributed systems.

Paper outline. The remainder of this paper is organized as follows. Section 7.1.1 describes related work and sets the context for the contributions of this paper. In Section 7.2, we discuss stack sharing and its consequences, and in Section 7.3 we present the system model that we use. Section 7.4 presents the exact formulation of determining the maximum stack usage, and gives the theoretical framework for Section 7.5, which describes algorithms for bounding the stack usage of systems with offsets and precedences. Section 7.6 gives an experimental evaluation of our analysis methods, and Section 7.7 concludes the paper and suggests future work.

7.1.1 Related work

A large number of publications have addressed preemption analysis for specific reasons, see, e.g. [2, 9, 15, 21, 22, 25]. Our work is related in the sense that we also investigate possible preemptions. However, our objectives differ, since

we analyze system wide premption patterns to investigate their effect on stack memory requirements for a task model with offsets and precedences.

Throughout the years, a number of publications have addressed stack sharing. Baker presented the Stack Resource Policy (SRP) that permits stack sharing among processes with shared resources [4]. Chatterjee *et al.* study stack boundedness for interrupt-driven programs [6]. In [8] Davis *et al.* address stack memory requirements and non-preemption groups to reduce shared stack usage. Gai *et al.* [11] present the Stack Resource Policy with preemption Thresholds (SRPT) which extends the work of Saksena and Wang [24] by accounting for stack usage when establishing non-preemption groups. In [12] Ghattas and Dean investigate stack space requirements under preemption threshold scheduling. Middha *et al.* [18] propose the MTSS stack sharing technique that allows a stack to grow into other tasks. In [23] Regehr *et al.* present a method to guarantee stack safety of interrupt-driven software by computing the worst-case memory requirements of individual interrupt handlers and perform preemption analysis between handlers. In [14] we presented an approximate stack analysis method to derive a safe upper bound on the shared stack usage of a static time-driven schedule in offset-based, hybrid scheduled (interrupt- and time-driven) fixed priority preemptive systems. In this paper, we extend that work by supporting stack sharing across several transactions for the task model with offsets. Here we also take precedence relations into account to further reduce possible preemptions.

7.2 Stack sharing in preemptive systems

In this paper we consider systems where several tasks use a single, statically allocated, run-time stack. For this to be possible, a task only uses the stack between the start time of an instance, v_i, and the finishing time of that instance, i.e., no data remains on the stack from one instance of a task to the next. This is ensured by not allowing tasks to suspend themselves voluntarily. In practice this means that OS-primitives like `sleep()` and `wait_for_event()` cannot be used. An invocation of a task can be viewed as a function call from the operating system, and the invocation terminates when the function call returns (thus any persistent context must be stored outside of the stack).

It is also required that a task instance never experiences blocking once it has started execution, i.e., we can never preempt the executing task because a needed resource is locked by a lower priority task. This is achieved by using an *early blocking* resource access protocol such as the immediate inheritance protocol [5] or the stack resource policy [4].

The motivation for allowing tasks to share a common stack is that this shared stack can be smaller than the sum of the individual stacks without jeopardizing the correctness of the application. Shared-stack analysis aims at (pre run-time) deriving a safe, but tight, approximation of the worst case (run-time) size of the shared stack. As long as the amount of memory statically allocated for the shared stack does not exceed this bound, the absence of stack overflow errors is guaranteed.

At any given point in time, the size of the shared stack equals the sum of the current stack usage for each active task instance. The maximum size of the shared stack thus depends on two factors: (i) the stack memory usage of each task instance, and (ii) the possible preemption patterns among tasks.

Due to the difficulties in determining the exact stack usage at every point in time for a task instance, shared-stack analysis methods typically assume that whenever a task is preempted, it is preempted at its maximum stack depth. We make the same assumption. Bounds on maximum stack usage for a given task can be derived by abstract interpretation using tools such as AbsInt [1] and Bound-T [26].

Previous traditional approaches to account for the second factor, i.e., the possible preemption patterns, is based on the fact that at most one task from each priority level (or preemption level, if these two concepts do not coincide) can be active at the same time. Thus, a simple and safe approach for bounding the maximum shared stack usage is to sum the maximum individual stack usage of tasks at each priority (or preemption) level. We call this approach SPL (Sum of all Priority Levels), as described by Davis *et al.* [8]. SPL uses the following function to calculate a stack usage bound:

$$\sum_{p \in \text{all priority levels}} \max\left(\{S_i : \tau_i \text{ has priority } p\}\right) \tag{7.1}$$

where S_i denotes the maximum stack usage of task τ_i.

However, this approach can be very pessimistic, since it assumes a worst-case situation where tasks with maximum stack usage from each priority level preempt each other in a nested fashion. In practice, this situation could be impossible to achieve because of factors such as release times, deadlines, precedence constraints, and other dependencies that affect when tasks can execute.

The analysis approach proposed in this paper reduces the pessimism of the traditional method by investigating the possible preemption patterns in more detail. We formally define the start- and finishing time of a task instance υ_i, as follows:

st_i The absolute time when υ_i actually begins its execution.

ft_i The absolute time when υ_i terminates its execution.

Assume that a task instance υ_i is preempted by another task instance υ_j. The use of an early-blocking resource protocol then ensures that $ft_j < ft_i$ if $st_i < st_j$, and the following holds:

$$st_i < st_j < ft_j < ft_i. \tag{7.2}$$

In this paper we are interested in chains of nested preemptions, so-called *preemption chains*. We define a preemption chain to be a sequence $PC = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ of task instances such that

$$st_1 < st_2 < \cdots < st_k < ft_k < ft_{k-1} < \cdots < ft_1. \tag{7.3}$$

Lemma 1. *$PC = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ is a preemption chain if and only if for all instances υ_i, υ_j in PC where $i < j$, it holds that $st_i < st_j < ft_j < ft_i$.*

Proof of Lemma 1. *The proof of Lemma 1 follows trivially from Equations (7.2) and (7.3).*

Let $AllPC$ be the set of all preemption chains in all runtime scenarios. Then, under the assumption that the worst case stack usage S_i of a task instance υ_i can occur at any time during its execution, a bound on the worst case stack usage SWC for a preemptive shared stack system can be expressed as follows:

$$SWC = \max_{PC \in AllPC} \sum_{\upsilon_i \in PC} S_i. \tag{7.4}$$

This formulation, however, cannot be directly used for analyzing and dimensioning the shared system stack since it is based on the dynamic (only available at run-time) properties st_i and ft_i. To be able to statically analyze the system, one has to relate the static task properties to these dynamic properties. This is done by establishing how the system model, scheduling policy, and run-time mechanism constrain the values of the actual start and finishing times.

In previous work we have described how this can be done for the special case that only tasks in the same transaction share stack [14]. This paper extends the analysis in the sense that we allow stack sharing among arbitrary transactions consisting of fixed priority tasks with offsets. We also improve the way precedence relations are accounted for in the preemption analysis.

7.3 System model

The system model used in this paper is an offset-based (or transactional) task model which was introduced by Tindell [27]. Later it was formally defined and extended by Gutiérrez and Harbour [13] and further improved upon by Mäki-Turja and Nolin [20]. Gutiérrez and Harbour show how this task model can be used to model precedence relations in a distributed system and perform a holistic schedulability analysis for the entire system [13]. In [14, 19] we show how it can be used to model hybrid, static and dynamic, scheduled systems, which is supported by a commercial operating system provided by Arcticus Systems [3].

The system model is defined as follows: the system, Γ, consists of a set of k transactions $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_s is activated by an event, and T_s denotes the minimum inter-arrival time between two consecutive events. The activating events can be mutually independent, i.e. the transactions may execute with arbitrary phasing. A transaction Γ_s contains $|\Gamma_s|$ tasks. A task may not be released for execution until a certain time (the *offset*) has elapsed after the arrival of the activating event.

We use τ_{si} to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the index of the task within the transaction. A task, τ_{si}, is defined by a worst-case execution time (C_{si}), an offset (O_{si}), a deadline (D_{si}), a maximum jitter (J_{si}), a maximum blocking from lower priority tasks (B_{si}), and a priority (P_{si}). S_{si} is used to denote the maximum stack usage of τ_{si}. When referring to the stack usage of a specific instance υ_j of a task τ_{si} we sometimes use S_j instead of S_{si} to simplify the presentation.

The system model is formally expressed as:

$$\begin{aligned}\Gamma :=& \{\langle \Gamma_1, T_1 \rangle, \ldots, \langle \Gamma_k, T_k \rangle\} \\ \Gamma_s :=& \{\tau_{s1}, \ldots, \tau_{s|\Gamma_s|}\} \\ \tau_{si} :=& \langle C_{si}, O_{si}, D_{si}, J_{si}, B_{si}, P_{si}, S_{si} \rangle\end{aligned}$$

Jitter is assumed to be a nonnegative variation in task release times, and the deadline of a task is relative to the triggering of the transaction. There are no other restrictions placed on deadline or jitter, i.e., they can each be either smaller or greater than the transaction period.

We assume that offsets are nonnegative and smaller than the period, that the system is schedulable, and that worst-case response times (R_{si}) have been calculated for all tasks [20]. Response times are measured from the triggering event of the transaction, i.e., including offset and jitter.

In addition, we define a binary non-preemption relation NOPRE between tasks such that if NOPRE(τ_{si}, τ_{tj}) then τ_{si} can not be preempted by τ_{tj}. The relation is reflexive, because two instances of the same task can never preempt each other. For the analysis in this paper, information about precedences between tasks in the system are taken into account by encoding these as non-preemption relations. Two tasks with a precedence relation between them will not preempt each other given that the response times of both tasks are less than the transaction period. Additional mutual exclusion information can, if available, also be encoded using NOPRE.

We assume the system is scheduled with fixed priority scheduling with fifo-scheduling of tasks with the same priority. Further, we assume that an early-blocking resource access protocol, such as the immediate inheritance protocol, is used.

7.4 Preemption analysis for offset-based systems

In the rest of the paper we assume that all tasks share a common stack. The upper bound problem for multiple transactions can then be informally stated as finding the maximum stack usage of all possible preemption chains, no matter in which transaction they occur.

Stack analysis for multiple transactions is naturally more complex than analysis of one single transaction, since tasks in different transactions may interfere in nontrivial ways depending on relative priorities and the phasing between transactions. To get a safe upper bound on the stack size we therefore need to examine all possible phasing patterns between transactions.

A straightforward approach for analyzing multiple transaction stack behavior is to analyse the transactions in isolation, using the sum for all transactions as an upper bound on the total stack consumption. Each transaction can be analyzed using the method developed in [14] or any other method. The result obtained is a safe upper bound if the analysis for each transaction is safe.

Unfortunately, the latter approach ignores that the global stack upper bound may be significantly lower, since all possible transaction-local preemption patterns may not occur at the same time. One example of this is when two stack-intensive tasks with equal priority both influence the stack bound in their respective transaction. The bound obtained can be pessimistic since no two tasks with equal priority can both be active at the same time.

In this section, we propose a new, more elaborate algorithm which takes this into account. The method is based on a precise analysis of the relaxed global precedence chains that are possible. The algorithm has a non-polynomial

time complexity but is nonetheless usable for analyzing realistically sized task sets. However, since sufficiently large task sets will never be analyzable using non-polynomial algorithms, we also propose a less accurate but still competitive approximate method with a polynomial time complexity. The method is a generalization of the one presented in [14] to handle several transactions.

7.4.1 Pairwise preemptions

Since task preemption is one of the factors influencing the size of the shared stack, a first step is to formulate a safe approximation of possible pairwise preemptions. For this, we first define the release time rt_i of a task instance υ_i as the absolute time when υ_i is ready to start executing.

Let α_k denote the activation time of the transaction releasing an instance υ_k of a task τ_{si}. Then, we have the following relations on the start time and release time of υ_k:

$$\alpha_k + O_{si} \leq st_k. \tag{7.5}$$

$$rt_k \leq \alpha_k + O_{si} + J_{si}. \tag{7.6}$$

These concepts are illustrated in Figure 7.1.

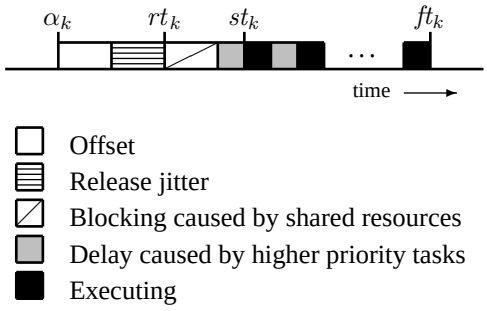


Figure 7.1: Important activities and time points for a task instance υ_k.

We use ψ_{sji} to denote the offset phasing between two tasks τ_{sj} and τ_{si} in the same transaction Γ_s, and define it as the minimum distance from an instance of τ_{sj} to the next instance of τ_{si}, or formally:

$$\psi_{sji} = (O_{si} - O_{sj}) \bmod T_s. \tag{7.7}$$

Generalizing the preemption criteria by Dobrin and Fohler [9], which is further extended in [14], we form the binary relation $\tau_{si} \prec \tau_{tj}$ with the interpretation

that τ_{si} may be preempted by τ_{tj}. We let the relation hold whenever (a) τ_{si} has lower priority than τ_{tj}, (b) τ_{si} does not have a non-preemption relation to τ_{tj}, and either (c1) τ_{si} and τ_{tj} are in different transactions (and can possibly intersect due to unknown phasing), (c2) τ_{tj} can be delayed by jitter, possibly starting after the next invocation of τ_{si}, or (c3) τ_{si} can possibly finish after the start of the next invocation of τ_{tj}. Formally, the relation can be defined as follows:

$$\begin{aligned} \tau_{si} \prec \tau_{tj} \equiv \overbrace{P_{si} < P_{tj}}^{\text{(a)}} \wedge \overbrace{\neg\text{NOPRE}(\tau_{si}, \tau_{tj})}^{\text{(b)}} \\ \wedge \quad \big(\underbrace{s \neq t}_{\text{(c1)}} \vee \underbrace{J_{sj} > \psi_{sji}}_{\text{(c2)}} \vee \underbrace{R_{si} - O_{si} > T_s - \psi_{sji}}_{\text{(c3)}} \big) \end{aligned} \tag{7.8}$$

Lemma 2. *The $\prec$ relation is a safe approximation of the possible preemptions between tasks. That is, if τ_{si} can under any run-time circumstance be preempted by τ_{tj}, then $\tau_{si} \prec \tau_{tj}$ holds.*

Proof of Lemma 2. *If an instance υ_k of τ_{si} is preempted by an instance υ_l of τ_{tj}, then we must have $P_{si} < P_{tj}$, $\neg\text{NOPRE}(\tau_{si}, \tau_{tj})$ and $st_k < st_l < ft_k$. From the assumption about the resource protocol, we know that τ_{si} can not start between rt_l and st_l, and thus we must have $st_k < rt_l$.*

If $s \neq t$, then $\tau_{si} \prec \tau_{tj}$ holds. Thus, for the remaining proof we assume $s = t$, and consider two cases:

***Case 1:** If $O_{si} < O_{sj}$, then $\psi_{sji} = T_s + O_{si} - O_{sj}$.*
If $\alpha_k \leq \alpha_l$, we have
$st_l < ft_k \;\Rightarrow\; \alpha_l + O_{sj} < \alpha_k + R_{si} \;\Rightarrow$
$O_{sj} - O_{si} < R_{si} - O_{si} \;\Rightarrow\; T_s - \psi_{sji} < R_{si} - O_{si}$.
If $\alpha_k > \alpha_l$, then $\alpha_l + T_s \leq \alpha_k$, and we have
$st_k < rt_l \;\Rightarrow\; \alpha_k + O_{si} < \alpha_l + O_{sj} + J_{sj} \;\Rightarrow$
$\alpha_l + T_s + O_{si} < \alpha_l + O_{sj} + J_{sj} \;\Rightarrow$
$T_s + O_{si} - O_{sj} < J_{sj} \;\Rightarrow\; \psi_{sji} < J_{sj}$.
***Case 2:** If $O_{si} \geq O_{sj}$, then $\psi_{sji} = O_{si} - O_{sj}$.*
If $\alpha_k \geq \alpha_l$, we have
$st_k < rt_l \;\Rightarrow\; \alpha_k + O_{si} < \alpha_l + O_{sj} + J_{sj} \;\Rightarrow$
$\alpha_l + O_{si} < \alpha_l + O_{sj} + J_{sj} \;\Rightarrow$
$O_{si} - O_{sj} < J_{sj} \;\Rightarrow\; \psi_{sji} < J_{sj}$.
If $\alpha_k < \alpha_l$, then $\alpha_k + T_s \leq \alpha_l$, and we have
$st_l < ft_k \;\Rightarrow\; \alpha_l + O_{sj} < \alpha_k + R_{si} \;\Rightarrow$
$\alpha_k + T_s + O_{sj} < \alpha_k + R_{si} \;\Rightarrow$

$T_s + O_{sj} - O_{si} < R_{si} - O_{si} \Rightarrow$
$T_s - \psi_{sji} < R_{si} - O_{si}$.
In all four subcases, we either have $T_s - \psi_{sji} < R_{si} - O_{si}$ *or* $\psi_{sji} < J_{sj}$*, which means that* $\tau_{si} \prec \tau_{tj}$ *holds.* □

7.4.2 Possible preemption chains

A sequence Q of tasks is a *possible preemption chain* (PPC) if it holds that $\tau_{si} \prec \tau_{tj}$ for all τ_{si}, τ_{tj} in Q where τ_{si} occurs before τ_{tj} in the sequence.

In other words, Q is a PPC if and only if the relation $\prec$ holds transitively between all tasks in Q. For example, the sequence $\{\tau_{11}, \tau_{12}, \tau_{13}\}$ is a PPC if and only if $\tau_{11} \prec \tau_{12}, \tau_{11} \prec \tau_{13}$ and $\tau_{12} \prec \tau_{13}$. If it only holds that $\tau_{11} \prec \tau_{12}$ and $\tau_{12} \prec \tau_{13}$, then Q is *not* a PPC.

The stack usage SU_Q of a PPC Q is the sum of the stack usage of the individual tasks in the chain, i.e., $SU_Q = \sum_{\tau_{si} \in Q} S_{si}$.

Lemma 3. *If* $PC = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ *is a preemption chain, and* $Q = \{\tau_{s_1 i_1}, \tau_{s_2 i_2}, \ldots, \tau_{s_k i_k}\}$ *is a corresponding sequence of tasks such that* $\upsilon_q \in PC$ *is an instance of* $\tau_{s_q i_q}$*, then* Q *is a PPC.*

Proof of Lemma 3. *For all task instances* υ_p, υ_q *in a preemption chain* PC*, if* $p < q$ *then it holds that* $st_p < st_q < ft_p$*. Since* υ_p *and* υ_q *are instances of* $\tau_{s_p i_p}$ *and* $\tau_{s_q i_q}$ *respectively, Lemma 2 implies that* $\tau_{s_p i_p} \prec \tau_{s_q i_q}$*, and thus* Q *is a PPC.* □

A PPC Q for which no other PPC has a higher stack usage in the same system is called a *maximal stack usage* PPC, or more informally, a *maximal* PPC. The stack upper bound problem can now be stated as finding a maximum stack usage PPC. We refer to this as the MAXPPC problem. We now prove that the stack usage of a maximal PPC Q in a system Γ is a safe upper bound on the stack usage of Γ.

Theorem 1. *The stack usage of a maximal PPC* Q *is a safe upper bound on the actual worst case stack usage for a system* Γ.

Proof of Theorem 2. *Let* Ψ *be the sequence of tasks instances participating in the preemption situation which cause the worst case stack usage, that is,* $SWC = \sum_{\tau_{si} \in \Psi} S_{si}$*. According to Lemma 3, we have that* Ψ *is a PPC with* $SU_\Psi = SWC$*. Since* Q *is a maximal PPC,* $SU_\Psi \leq SU_Q$*, which concludes the proof.* □

7.5 Algorithms

In [14], we proposed a polynomial method for stack analysis of a single transaction of the type described in Section 7.3. The polynomial time behavior of this method comes from the fact that a single transaction represented by tasks with offsets and response times can be efficiently analyzed using specialized graph algorithms [16]. These algorithms cannot be directly applied to analysis of a global stack shared by several transactions. When analyzing a single transaction in isolation, the task offsets and response times can be used to bound the time interval within which the tasks will execute. However, when several transactions are considered, we no longer have a common activation time, and therefore the graph algorithms used in [14] are no longer applicable. We therefore propose to analyze smaller systems using an exact algorithm, guaranteed to find a maximal PPC. For larger systems, we propose to use a polynomial approximation, described in Section 7.5.4.

7.5.1 An exact algorithm for the MAXPPC problem

We solve the problem of finding a maximal PPC by forming a (directed) *preemption graph* of nodes representing tasks, and edges representing possible preemptions, as defined in (7.8). An example taskset (assuming $J = B = 0$ and $\neg\text{NOPRE}(\tau_{si}, \tau_{tj})$ for all tasks) and its corresponding preemption graph is shown in Figure 7.2, where solid edges represent possible preemptions within a transaction, and dashed edges represent possible preemptions between different transactions.

Task	O	P	R	S
τ_{11}	0	1	3854	4
τ_{12}	1697	2	3837	1
τ_{13}	4635	4	4781	2
τ_{21}	0	3	393	2
τ_{22}	617	1	3854	1
τ_{23}	2588	3	3699	3

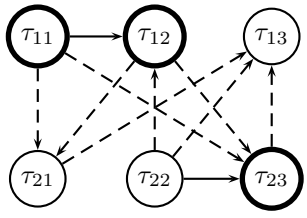


Figure 7.2: An example of a preemption graph, also showing tasks in a maximal PPC.

The preemption graph is not necessarily transitive, as Figure 7.2 shows. This implies that not all paths in the preemption graph form PPCs. As an ex-

ample, the path $Q = \{\tau_{11}, \tau_{12}, \tau_{23}, \tau_{13}\}$ is not a PPC since several preemption relations, for example the one between τ_{12} and τ_{13}, are missing.

The method is based on a branch-and-bound search for PPCs in this graph, recursively building PPCs Q^i. An outline of the algorithm is given in Algorithm 7.1. Initially, $Q^0 = M = \emptyset$. If in any recursive step the total stack usage SU_{Q^i} is greater than the stack usage SU_M of the current maximal PPC M, then Q^i becomes the new maximum. We define a *cover set* $C(Q)$ of a PPC Q as a set of tasks for which all tasks in $C(Q)$ can possibly preempt all tasks in Q. A cover set is *maximal* if it cannot be extended by any other task. The algorithm maintains $C(Q^i)$ and then recursively examines an extension $Q^{i+1} = Q^i \cup \{\tau_{tj}\}$ of Q^i for each task τ_{tj} in $C(Q^i)$. We also apply a bounding function UB to terminate search in branches which clearly cannot contain a maximum PPC. This bounding function is further discussed in Section 7.5.2.

Algorithm 7.1: Computing a maximal PPC in a generic preemption graph.

MAXPPC(Q)
(1) **if** $SU_Q > SU_M$ **then** $M \leftarrow Q$
(2) $C(Q) = \{\tau_{tj} \mid \forall \tau_{si} \in Q . \tau_{si} \prec \tau_{tj}\}$
(3) **if** $SU_Q + \text{UB}(C(Q)) \leq SU_M$ **then return**
(4) **foreach** $\tau_{tj} \in C(Q)$
(5) MAXPPC($Q \cup \{\tau_{tj}\}$)

In Section 7.5.3, we show that the *algorithm* described is *exact* in the sense that it always computes the maximal PPC, and therefore solves the MAXPPC problem. We also show that the method is *safe*, because the stack usage of the maximal PPC is an upper bound on the stack usage of the system.

Note that our method of *stack bounding* is not exact, since the $\prec$ relation is in itself a (safe) approximation. Also, there are other factors unaccounted for. For example, there may be further restrictions on the possible nesting patterns due to mutual exclusion, and the tasks may not use their maximum stack when preempted.

7.5.2 Bounding the maximal PPC

Choosing the right function for the bounding step in Algorithm 7.1 is essential to guarantee correctness and to acquire a method usable in practice. A *safe upper bound function* on the maximal PPC stack usage for a set of tasks C is a function UB for which $\text{UB}(C) \geq SU_K$, where $K \subseteq C$ is a maximal PPC. We use the most stack-intensive path in the preemption graph spanned by $C(Q)$ as the UB function, which we refer to as the PUB method. A heaviest path (w.r.t.

stack space) in a directed acyclic graph can be found in $O(n+m)$ time, where n is the number of vertices and m is the number of edges [7].

Theorem 2. *PUB is a safe upper bound function on the maximal PPC stack usage.*

Proof of Theorem 3. *From the definition of a PPC in Section 7.4.2, we have that a maximal PPC $K \subseteq C$ is a path with stack usage SU_K. PUB results in the maximum stack usage of any path $A \subseteq C$. Therefore, $PUB(C) \geq SU_K$, which concludes the proof.* □

We refer to the combination of the branch-and-bound method in Algorithm 7.1 with the most stack-intensive path relaxation (PUB) as bounding function, as the PPCBB algorithm.

7.5.3 Correctness

In order to claim correctness of Algorithm 7.1 we need to show that it computes a maximal PPC. Theorem 1 then gives us that the stack usage of the PPC computed by Algorithm 7.1 is an upper bound on the stack usage of the system. We first need to prove a lemma regarding the stack usage of a PPC when extended with tasks from a cover set.

Lemma 4. *If Q is a PPC, $C(Q)$ is a cover set of Q, and $K \subseteq C(Q)$ is another PPC, then $Q \cup K$ is a PPC with stack usage $SU_{Q\cup K} = SU_Q + SU_K$.*

Proof of Lemma 4. *All tasks in Q can be preempted by all tasks in $C(Q)$, and since Q and K are both PPCs and $K \subseteq C(Q)$, $Q \cup K$ is a PPC. Furthermore, $Q \cap C(Q) = \emptyset$ because no task can preempt itself, and thus $Q \cap K = \emptyset$, and $SU_{Q\cup K} = \sum_{\tau_{si}\in Q\cup K} S_{si} = \sum_{\tau_{si}\in Q} S_{si} + \sum_{\tau_{tj}\in K} S_{tj} = SU_Q + SU_K$.* □

We can now prove that Algorithm 7.1 is correct.

Theorem 3. *If* UB *is a safe stack usage upper bound function, then Algorithm 7.1 terminates with a maximal PPC.*

Proof of Theorem 4. *The proof is given in two parts.*

We first assume that $\mathrm{UB}(C) = \infty$ for all inputs C, so that Algorithm 7.1 never returns on line (3). Given a PPC Q and any task τ_{tj} from a maximal cover set $C(Q)$, we can form a new set $Q' = Q \cup \{\tau_{tj}\}$ which is also a PPC (from Lemma 4). Therefore, Q is always a PPC, and since the algorithm extends Q with one task from $C(Q)$ and $Q \cap C(Q) = \emptyset$, the algorithm will

terminate. If τ_{si} is not in $C(Q)$, then $Q \cup \{\tau_{si}\}$ is not a PPC. All together, the algorithm explores all PPCs, including a maximal PPC which will be stored in M and consequently returned when the algorithm terminates.

Now assume that UB *is a safe upper bound function on the maximal PPC stack usage in a set of tasks. From Lemma 4 we have $SU_{Q\cup K} = SU_Q + SU_K$ for all PPCs $K \subseteq C(Q)$, and subsequently this also holds if K is a maximal PPC in $C(Q)$, in which case $Q \cup K$ is also a maximal PPC in $Q \cup C(Q)$ (from the definition of cover set). Since* UB *is safe, $SU_Q + \text{UB}(C(Q)) \geq SU_{Q\cup K}$. Thus, if $SU_Q + \text{UB}(C(Q)) \leq SU_M$ where M is the most stack-intensive PPC found so far, there is no PPC in $Q \cup C(Q)$ which has a higher stack usage than M, and we can return from this branch without losing any maximal solutions.* □

7.5.4 Polynomial approximations

Algorithm 7.1 is non-polynomial. In Section 7.6, we show that despite this, the algorithm can be used to analyze realistically sized task-sets. However, an exponential analysis method will still be too time-consuming for practical use when the number of tasks under analysis is too large. We therefore propose a polynomial time approximation for analyzing stack size where the number of tasks is too high to be analyzed using the branch-and-bound method. The approximation is a combination of two methods. The first one, STLA, is based on analysis of individual transactions in isolation, and essentially uses the sum for all transactions as an upper bound on the total stack consumption. The method is described in [14], but has been modified for the current task model, to account for precedence constraints and to allow response times larger than the period. STLA is a safe upper bound if the analysis for each transaction is safe, and runs in $O(kn^3)$ time, where k is the number of transactions, and n is the maximal number of tasks in a single transaction.

STLA is overly pessimistic in situations where equally prioritized stack-intensive tasks in different transactions influence the isolated transaction stack upper bound. Since the tasks have equal priority, they cannot preempt each other, and the global upper bound obtained is pessimistic. To remedy this, we also use a second polynomial method to obtain a different upper bound. The method, called PUB, finds a maximum stack usage path in the global preemption graph of all tasks in the system, and is the same one described in Section 7.5.2 and used as an upper bound function in PPCBB.

To illustrate the difference between PPCBB, STLA and PUB, consider the task set illustrated in Figure 7.2. The maximal PPC in this task set is

$\{\tau_{11}, \tau_{12}, \tau_{23}\}$ with a total stack usage of 8. This is the result that PPCBB would return. In contrast, STLA would compute an upper bound by considering the two transactions $\Gamma_1 = \{\tau_{11}, \tau_{12}, \tau_{13}\}$ and $\Gamma_2 = \{\tau_{21}, \tau_{22}, \tau_{23}\}$ in isolation, computing the PPCs $\{\tau_{11}, \tau_{12}\}$ with stack usage 5 for Γ_1 and $\{\tau_{22}, \tau_{23}\}$ for Γ_2 with a stack usage of 4. The sum, 9, would be returned as the result. Finally, PUB would return the stack usage 10 of the most stack intensive path $\{\tau_{11}, \tau_{12}, \tau_{23}, \tau_{13}\}$ in the graph, which is not a PPC but is nonetheless, as shown in the proofs of Theorem 1 and 2, a safe approximation on the stack usage of the system.

Since $\tau_{si} \prec \tau_{tj} \rightarrow P_{si} < P_{tj}$, a stack usage path P can never include two tasks on the same priority level. If we would relax the $\prec$ relation into $\prec' \equiv P_{si} < P_{tj}$, the stack usage of the most stack intensive path would be equal to the maximum stack for each priority level in the system. Therefore, PUB is always at least as good as the traditional approach (SPL). We propose to use the minimum of PUB and STLA (referred to as STLA_PUB) as a polynomial time alternative to PPCBB. Since both PUB and STLA are safe, STLA_PUB is also safe.

7.6 Evaluation

We evaluate the efficiency of our proposed methods by generating random task sets and calculating the stack upper bounds. All tasks in each generated task set share one common stack. We use three methods (SPL, STLA_PUB, PPCBB) to calculate an upper bound on the shared system stack. Thus, the upper bounds are illustrated by the following graphs:

SPL: The traditional approach to determine an upper bound (sum of maximum stack usage of each priority/preemption level).

STLA_PUB: This represents the minimum of the polynomial methods STLA (analysis of individual transactions) and PUB (longest path in global preemption graph). See Section 7.5.4 for details.

PPCBB: Non polynomial branch-and-bound based method with longest path relaxation. See Section 7.5.1 and 7.5.2 for details.

7.6.1 Simulation setup

We run the stack analysis application on an Intel Pentium 4, 2.18 GHz with 512 MB of RAM. We generate random task sets as input to the stack analysis application. The task generator takes the following input parameters (default

values represent the base configuration of each analysis):

Parameter	**Default**
Number of transactions	5
Number of tasks	60
Total system load	40%
Task priority (min–max)	1–32
Task stack usage (min–max)	128–2048 bytes
Probability of precedence	10%

Using these parameters, task sets with the following characteristics are generated:

- The period time T_s of each transaction is set to 10,000.
- Each task offset (O_{si}) is randomly and uniformly distributed between 0 and $T_s/2$.
- Task priorities and the stack usages are assigned randomly between minimum and maximum value with a uniform distribution.
- The total system load, and the number of tasks in the system, is distributed among the transactions in such way that all transactions have the same amount of load and the same number of tasks.
- Worst case execution times, C_{si}, are initially assigned to each task in such way that tasks are separated in time within a transaction. The execution times are then adjusted by a fraction, so that the the total system load (as defined by the input parameter) is obtained, preserving time separation of tasks within a transaction.
- Each task is assigned a precedence relation with a probability specified by the precedence probability attribute. For example, if the probability of precedence for τ_{si} is 10%, then for each succeeding task (i.e., task with larger or equal offset than τ_{si}) in Γ_s, there is a 10% probability that τ_{si} is given precedence over the task. When all precedences are assigned, transitive precedences are established, e.g, if τ_{si} has precedence over τ_{sj} and τ_{sj} has precedence over τ_{sk}, then τ_{si} has precedence over τ_{sk}.
- We assign deadline $D_{si} = T_s$ for each task. All tasks are required to meet their deadlines (otherwise the task set is considered unschedulable). In case the generated task set is unschedulable, the task set is discarded and a new task set is generated.

7.6.2 Results

Each point in the graphs represents the mean stack usage of 100 randomly generated schedulable task sets. For each point, a confidence interval (confidence level of 95%) is shown to indicate the reliability of the figures. For each diagram, we vary one parameter, keeping all other parameters according to the base configuration. In addition to calculating upper bounds, we also measured the mean execution time for each method. The mean execution times for SPL lies in the range of micro seconds, for STLA_PUB the mean execution time lies in the in the range of milliseconds and for PPCBB in the range from milliseconds up to five seconds.

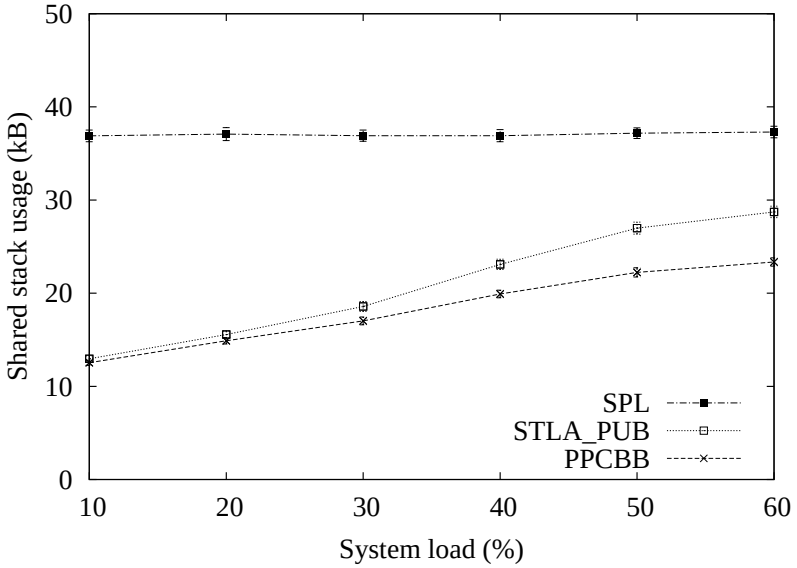


Figure 7.3: Varying system load.

In Figure 7.3 we vary the total system load from 10% to 60%. As expected, the stack upper bound using the traditional method (SPL) is constant and unaffected by variations in load. This is due to the fact that SPL only considers priorities when calculating the upper bound. Both STLA_PUB and PPCBB produces upper bounds that are slowly increasing with increasing load. This is natural, since increasing the load, keeping all other parameters according to the base configuration, typically results in larger response times, which in turn

increases the number of possible preemptions in the system.

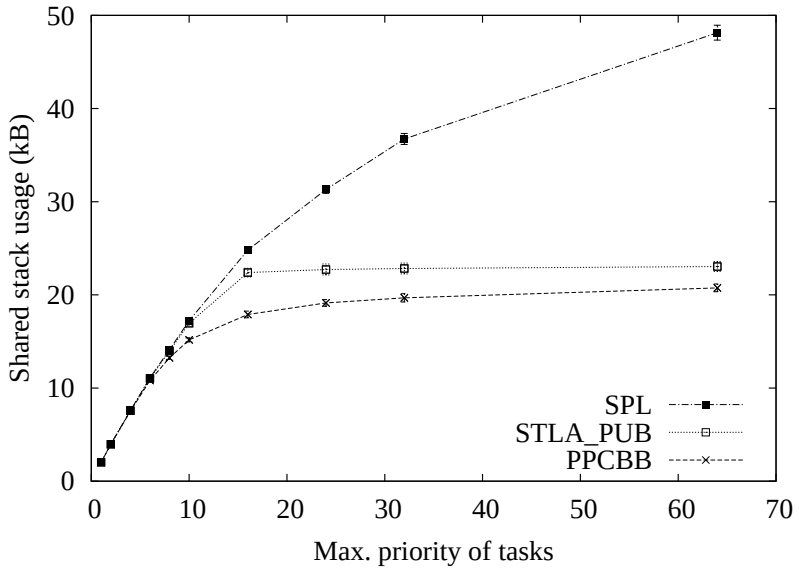


Figure 7.4: Varying maximum priority.

In Figure 7.4 we vary the maximum priority of tasks from 1 to 64. This gives a possible priority distribution of 1 to n, where n is indicated by the x-axis. We observe that for small values on n the difference between the methods is small. For larger values on n the difference is significant.

In Figure 7.5 we vary the number of tasks in the system from 10 to 100. With a low number of tasks in the system, there is a larger possibility that tasks have unique priorities hence considered to be part of a preemption chain by SPL. SPLA_PUB and PPCBB goes one step further and examines preemption patterns, with a tighter upper bound as a result.

In Figure 7.6 we vary the number of transactions from 1 to 20. We see that both SPLA_PUB and PPCBB increase when increasing the number of transactions in the system. With more transactions, the arbitrary phasing between them increases the possibility of nested preemptions, resulting in increased shared stack usage. SPL is constant and unaffected by variation in the number of transactions. Again, this is expected, since SPL only considers priorities.

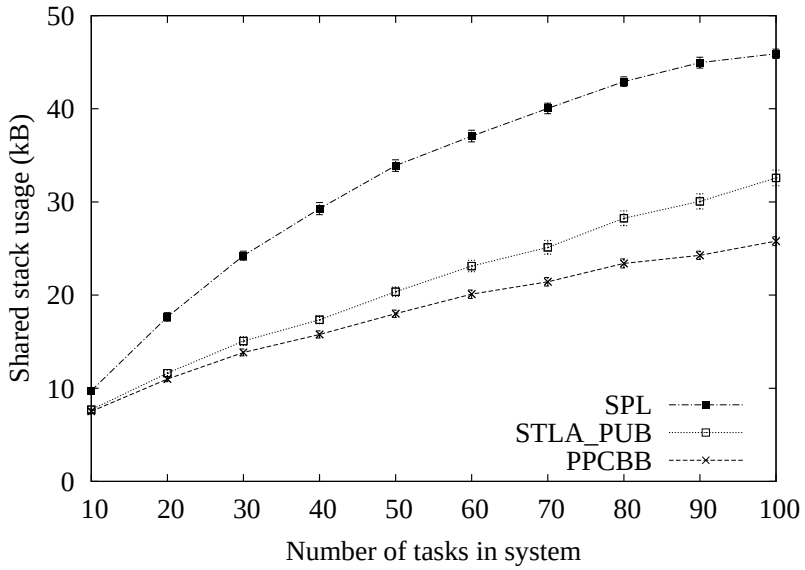


Figure 7.5: Varying the number of tasks in the system.

7.7 Conclusions and future work

Allowing tasks to share a common run-time stack can reduce the amount of RAM needed for an application. However, in order to safely reduce the overall run-time stack, one must be able to analyze possible preemption patterns statically. And from those preemption patters deduce the possible stack-usage. Static information about system model, scheduling policy and run-time mechanism can be used to constrain the values of the dynamic task-properties that affect possible preemptions, and thus also shared stack usage.

A task model with such static information is the task model with offsets (the transactional task model) where priorities, offsets and precedences limit the possible preemption patterns. We have, for that task model, developed a system wide preemption analysis that safely approximates the actual preemptions and forms a basis for safe upper bound of the total shared stack usage.

We presented two novel methods for determining a safe upper bound on the stack usage. Both methods analyze a graph consisting of tasks and possible preemptions between these. The first method is an exact search for maximal possible preemption chains. The second method is a combination of two algo-

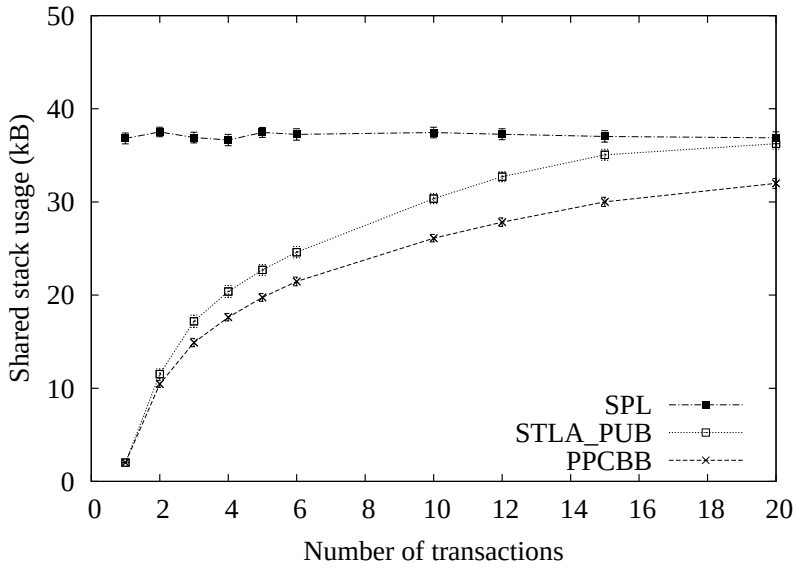


Figure 7.6: Varying the number of transactions.

rithms, both being polynomial approximations of the first. We formally showed that both methods are safe in the sense that they will never underestimate the amount of stack space needed.

The methods have a clear practical value in a verification/validation phase of system development. They can be used to formally verify that stack overflow will not occur during run time. In a simulation study, we evaluated our techniques and compared it to the traditional method to estimate stack usage. We found that our exact method significantly reduced the amount of stack memory needed. In our simulations, a decrease in the order of 40% was typical, with a runtime in the order of seconds. Our polynomial approximation consequently yields about 20% higher bound compared to the exact method.

In future work our methods can be used to further reduce the stack bound by more detailed modeling of the system behavior. For example, the assumption that each task uses its maximum stack when preempted may lead to overly pessimistic result if the stack usage is highly variable during execution. With knowledge about the variation of stack usage, one might split a task into several segments, each with its own stack usage. These segments can then be

modeled as separate tasks with precedence constraints, and thus we should obtain a tighter bound on the stack usage. Furthermore, a similar technique could also be used to split up a task that uses shared resources where the part that uses the resource can be modeled as a task with a mutual exclusion relation to other tasks that uses the same resource.

Acknowledgement

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

Bibliography

[1] *AbsInt*. Web page,
http://www.absint.com/stackanalyzer/.

[2] J. H. ANDERSON, S. RAMAMURTHY, AND K. JEFFAY, *Real-Time Computing with Lock-Free Shared Objects*, ACM Transactions on Computer Systems, 15 (1997), pp. 134–165.

[3] *Arcticus Systems*. Web page,
http://www.arcticus-systems.com.

[4] T. P. BAKER, *A Stack Based Resource Allocation Policy for Real-Time Processes*, in Proceedings of the 11th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1990.

[5] A. BURNS AND A. WELLINGS, *Real-Time Systems and Programming Languages*, Addison-Wesley, second ed., 1996, ch. 13.10.1 Immediate Ceiling Priority Inheritance.

[6] K. CHATTERJEE, D. MA, R. MAJUMDAR, T. ZHAO, T. HENZINGER, AND J. PALSBERG, *Stack size analysis for interrupt-driven programs*, in Proceedings of the 10th Annual International Static Analysis Symposium, June 2003.

[7] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, MIT Press, Cambridge, MA, USA, second ed., 2001.

[8] R. DAVIS, N. MERRIAM, AND N. TRACEY, *How Embedded Applications using an RTOS can stay within On-chip Memory Limits*, in Proceedings of the Work-in-Progress and Industrial Experience Session, Euromicro Conference on Real-Time Systems, June 2000.

[9] R. DOBRIN AND G. FOHLER, *Reducing the Number of Preemptions in Fixed Priority Scheduling*, in Proceedings of the 16th Euromicro Conference on Real-time Systems, IEEE Computer Society, 2004.

[10] *Evidence Srl.* Web page, http://www.evidence.eu.com.

[11] P. GAI, G. LIPARI, AND M. D. NATALE, *Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip*, in Proceedings of the 22th IEEE Real-Time Systems Symposium, IEEE Computer Society, 2001.

[12] R. GHATTAS AND A. DEAN, *Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis*, in Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2007.

[13] J. C. P. GUTIERREZ AND M. G. HARBOUR, *Schedulability Analysis for Tasks with Static and Dynamic Offsets*, in Proceedings of the 19th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1998.

[14] K. HÄNNINEN, J. MÄKI-TURJA, M. BOHLIN, J. CARLSON, AND M. NOLIN, *Determining Maximum Stack Usage in Preemptive Shared Stack Systems*, in Proceedings of the 27th IEEE Real-Time Systems Symposium, IEEE Computer Society, 2006.

[15] C. G. LEE, K. LEE, J. HAHN, Y. M. SEO, S. L. MIN, R. HA, S. HONG, C. Y. PARK, M. LEE, AND C. S. KIM, *Bounding Cache-Related Preemption Delay for Real-Time Systems*, IEEE Transactions on Software Engineering, 27 (2001), pp. 805–826.

[16] T. A. MCKEE AND F. MCMORRIS, *Topics in Intersection Graph Theory*, no. QA 166.105.M34 in SIAM Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, 1999.

[17] *Micro Digital Inc.: smx® Special Features*. Available 2009-10-20 at: http://www.smxinfo.com/rtos/kernel/smxfeatr.pdf.

[18] B. MIDDHA, M. SIMPSON, AND R. BARUA, *MTSS: Multi Task Stack Sharing for Embedded Systems*, in Proceedings of the of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, San Francisco, CA, Sept 2005.

[19] J. MÄKI-TURJA, K. HÄNNINEN, AND M. NOLIN, *Efficient Development of Real-Time Systems Using Hybrid Scheduling*, in International Conference on Embedded Systems and Applications, June 2005.

[20] J. MÄKI-TURJA AND M. NOLIN, *Fast and tight response-times for tasks with offsets*.

[21] H. RAMAPRASAD AND F. MUELLER, *Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks*, in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2006.

[22] J. REGEHR, *Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults*, in Proceedings of the 23rd IEEE Real-Time Systems Symposium, IEEE Computer Society, 2002.

[23] J. REGEHR, A. REID, AND K. WEBB, *Eliminating Stack Overflow by Abstract Interpretation*, ACM Transactions in Embedded Computing Systems, 4 (2005), pp. 751–778.

[24] M. SAKSENA AND Y. WANG, *Scalable Real-Time System Design Using Preemption Thresholds*, in Proceedings of the 21st IEEE Real-Time Systems Symposium, IEEE Computer Society, 2000.

[25] J. STASCHULAT, S. SCHLIECKER, AND R. ERNST, *Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay*, in Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2005.

[26] *Tidorum*. Web page, http://www.tidorum.fi/bound-t/.

[27] K. TINDELL, *Using Offset Information to Analyse Static Priority Preemptively Scheduled Task Sets*, Tech. Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[28] *Unicoi Systems*. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.

Chapter 8

Paper C: Best-Effort Simulation-Based Timing Analysis using Hill-Climbing with Random Restarts

Markus Bohlin, Yue Lu, Johan Kraft, Per Kreuger and Thomas Nolte.

In Proceedings of the 15th International Conference on Real-time and Embedded Computing Systems and Applications.

August 24–26, 2009, Beijing, People's Republic of China.

Abstract

Today, many companies developing real-time systems have no means for accurate timing analysis, as the software violates the assumptions of traditional analytical methods for response-time analysis, and are too complex for exhaustive analysis using e.g. model checking. This paper presents an efficient best-effort approach for timing analysis targeting such systems, where simulations of a detailed system model are controlled by a simple yet novel optimization algorithm, based on hill climbing with random restarts (HCRR). Using a simulation-based approach implies that the result is not guaranteed to be the worst-case response time, but on the other hand, the method can handle in principle any software design. Unlike previous approaches, the new algorithm directly manipulates simulation parameters such as execution times, arrival jitter and input stimulus.

A thorough evaluation is also presented, where HCRR is compared to Monte Carlo simulation (the current state-of-practice) and a previously proposed method. The evaluation is performed using a set of simulation models constructed from existing systems in the robotics and vehicular domain, and shows that for the three models investigated, the proposed method was 4-11 % more accurate and vastly more efficient than the other methods. In our evaluation, HCRR found the second-best result on average 42 times faster than the second-best method. For the largest model, HCRR used only 7.6 % of the simulations needed by the second-best method to reach the same result, implying that HCRR scales to larger systems. For the most realistic model, our new method found the highest-known response time 1628 times faster than the second-best method.

8.1 Introduction

Today, most existing embedded real-time systems have been developed in a traditional code-oriented manner. Many of them are also maintained over extended periods of time, sometimes spanning decades, during which they become larger and more complex due to the iterative changes made as part of the system evolution and maintenance. The increasing complexity makes these systems increasingly hard and expensive to maintain and verify.

One specific problem with such systems is the risk for introducing timing-related errors. A natural approach to avoid timing-related errors in real-time software would be to use established analytical methods for response-time analysis (RTA, [10, 15]), which provides exact worst-case response times of tasks, given correct worst-case execution times (WCET). Thereby, a system's correctness with respect to temporal requirements can be guaranteed.

The ability to perform timing analysis, using RTA or other means, does not only improve the quality of the system verification, but can also reduce development and maintenance costs significantly as potential timing-related errors can be identified early, during the design of new features, and thereby avoided. Timing errors can otherwise only be detected in late verification phases, where detected bugs often case major costs and delays. Moreover, timing errors often only occur under very specific conditions, which are hard to detect using testing.

Sadly, it is not possible to make practical use of RTA on a large quantity of existing industrial software systems, as they violate the assumptions of the method. Such systems might have been initially designed without timing analysis in mind, or development personnel may have introduced violations of RTA during the system evolution, and thereby lost the analyzability.

The authors have observed several issues with respect to RTA in existing industrial/embedded software systems. Some relevant examples are:

- Tasks communicate and trigger other tasks in complex undocumented patterns.
- Task WCET often depends on input.
- The task priority is sometimes changed dynamically.
- Deadlines are not always defined explicitly, but manifest as functional errors when different timeouts expire.

Moreover, some implementations observed in industrial code makes it very

hard to perform static WCET analysis. Consider the following example, where a task reads all messages in a message queue and process them accordingly:

```
do {
   msg = receiveMessage(MyMessageQueue);
   process_msg(msg);
} while (msg != NO_MESSAGE);
```

There are at least two aspects in this example which are hard to analyse statically. First, even though the maximum allowed queue size is usually known, the actual maximum at runtime is not; the developers may have over-dimensioned the queue as a safety margin. Second, and more importantly, other tasks may preempt the execution of the loop and refill the queue. When this happens, the number of loop iterations is no longer bounded by the maximum queue size.

The impact of mechanisms like buffered queues and priority changes can cause very intricate scenarios, where the worst-case is counter-intuitive and extremely hard to predict manually. For instance, in one of the simulation models (Model 1, described in Section 8.4.1) used for evaluation of our approach, the worst-case response time of the task in focus surprisingly did not occur when the model received the maximum amount of input events. Instead, the worst case turned out to occur when the input events formed a completely different and very intricate pattern. The details of this case is described in [13]. The system model used by analytical methods such as RTA is too simplistic to allow accurate timing analysis of such systems with such behaviors, instead a detailed model is required, where also relevant task behavior can be described.

An example of an industrial real-time system where RTA is not applicable is the control system for industrial robots developed by ABB. This system has a very complex temporal behavior, where some tasks have execution times varying radically due to input-dependent IPC and globally shared state variables, and where tasks may even change scheduling priority. The analytical methods' use of a task-level WCET attribute will in such cases be very pessimistic since the tasks are not independent; there are often dependencies which result in mutual exclusion between different tasks' WCET scenario.

A more detailed system model is therefore necessary for timing analysis of such systems. Ideally, the model should describe the detailed execution control flow on a code level with respect to resource usage and interaction, e.g., inter-process communication, CPU time and logical resources. Simulation-based methods has previously been shown to work well in analysing such large and detailed models, since they only sample the system state space rather than

attempting to search it exhaustively. Moreover, simulation-based analysis is far more efficient in finding potential timing problems than system-level testing, the dominating method in industry today. Several frameworks already exist for timing simulation of real-time system models, e.g., the commercial tool *VirtualTime* [19] and the academic tool *ARTISST* [7]. These solutions rely on Monte Carlo simulation, which can be described as keeping the highest results from a set of randomized simulations.

In this paper, we show that a detailed representation of the simulation parameters in combination with a focused optimization algorithm can yield substantially better results than both Monte Carlo simulation (which is the current state-of-practice) and another previously proposed method, *MABERA* [14]. Specifically, we propose a new approach where key aspects of the system at hand are encoded directly as parameters in the algorithm. We then use a fairly straightforward optimization method based on the well-known hill-climbing algorithm [20]. Surprisingly enough, nobody seems to have tried this before.

The paper contains the following contributions: 1) We give an explicit representation of simulation instances in the form of inputs such as execution time, arrival jitter and external input stimulus is defined, 2) we present a novel algorithm for manipulating simulation parameters, based on the simple idea of hill-climbing with random restarts (HCRR), and 3) we give a thorough experimental evaluation of performance, scaling and convergence of the new algorithm, comparing the results to those obtained from MABERA and Monte Carlo simulation. In the evaluation, we show that the new algorithm is significantly better than previous approaches in identifying extreme response times using a limited number of simulations.

The paper outline is as follows. Section 8.2 presents related work and the new input representation. Section 8.3 presents the new approach proposed in this paper, and Section 8.4 describes a set of case-study models used to evaluate the approach. The evaluation is presented in Section 8.5, and finally, Section 8.6 concludes.

8.2 Best-Effort Response-Time Analysis

Response-time analysis is certainly not something new, and besides the standard approaches such as RTA [10, 15], formal analysis tools like UPPAAL [6, 22] can also be used for exhaustive analysis of software systems, but for industrial-sized models, the state space can grow too large for them to be practically useful.

The use of evolutionary algorithms for different types of test case gener-

ation has also been studied for quite some time. In [2], genetic algorithms were used to generate test cases for a software relay system used in electrical networks. The purpose of the genetic algorithm is to provoke high response times for the software, which executed in a simulation environment. Nossal et al [17] describe various extensions of the traditional genetic algorithm [9] to better suit the type of problems in the real-time domain. More recently, Mueller and Wegener [16] gave a comprehensive comparison of static analysis techniques and evolutionary algorithms, with regard to schedulability, for several real-time applications.

In [21], Samii et al aim to find extreme response times for distributed systems by optimizing a set of simulation parameters for models containing temporal attributes and communication. They use a genetic algorithm to explore combinations of task execution times in order to maximize end-to-end response time. Flow of control within tasks is not considered. Their results depend on the method developed by Racu and Ernst [18] for identifying situations where decreased execution times can lead to increased response times. The analysis framework by Kim et al [11] also has a similar basis of temporal task attributes.

In [14], we presented MABERA, a meta-heuristic approach for best-effort response-time analysis of models of complex legacy systems using ideas from genetic algorithms [9].

The approach is based on a simulator using a schedule of random number generator seeds, in turn used to generate random numbers for the parameters of the adhering system model. The seed of the random number generator can be changed at arbitrary time points, and thus provide a crude control mechanism. Due to the seed schedule representation, only mutation is used in the evolutionary algorithm, which inserts randomly selected new seeds at specific simulation time points. The effect of seed switching is that the entire execution trace for the rest of the simulation is changed. Unfortunately this implies that it is not possible to modify a restricted subset of the simulation parameters, for example the execution time for a specific code segment, that might on its own severely affect the response time. For heuristic methods to work well, small changes in a candidate solution should have small but noticeable effects on the objective function. This clearly doesn't hold for MABERA, where a newly inserted seed makes the rest of the simulation behave completely different. Readers can refer to [14] for a more thorough description of the MABERA approach.

8.2.1 Simulation of Complex Real-Time Systems

The analysis method presented in this paper is based on the simulator framework *RTSSim*, [13], which allows for simulating models describing both the functional and temporal behaviour of tasks. An RTSSim simulation model consists of a set of tasks, sharing a single processor. Each task in RTSSim is a C program, which executes in a "sandbox" environment with similar services and runtime mechanisms as a normal real-time operating system, e.g., task scheduling, inter-process communication (message queues) and synchronization (semaphores). The scheduling policy or RTSSim is preemptive fixed-priority scheduling and each task has scheduling attributes such as priority, periodicity and offset. It is possible to change these parameters dynamically, in the task model code, to implement a custom scheduling policy, on top of the default scheduling policy.

In RTSSim, time is represented in a discrete manner using an integer simulation clock, which is only advanced explicitly by the tasks in the simulation model, using a special routine, EXECUTE. Calls to this routine models the tasks' consumption of CPU time.

All time-related operations in RTSSim, such as timeouts and activation of time-triggered tasks, are driven by the simulation clock, which makes the simulation result independent of process scheduling and performance of the simulation computer. The response time of tasks is measured whenever the scheduler is invoked, which happens for example at IPC, task switches, EXECUTE statements, operations on semaphores, task activations and when tasks end. This, together with the simulation clock behaviour, guarantees that the measured response time is exact.

The simulation framework allows for three types of selections which are directly controlled by simulator input data.

1. selection of execution times (for EXECUTE),
2. selection of task-arrival jitter, and
3. selection of task control flow.

A simulation in RTSSim is completely deterministic given a specific input, in this paper referred to as a *simulation instance*. Monte Carlo simulation is realized by providing a randomly generated simulation instance.

The models used for the evaluation in this paper were manually designed to contain similar modeling and analysis challenges as the real systems, and

contain only the aspects which were considered interesting from a timing analysis perspective. In general, however, a major issue when using simulation for analysis of existing systems is how to obtain the necessary simulation model, which should be a subset of the original program focusing on behaviour of significance for task scheduling, communication and allocation of logical resources. For many systems, manual modeling would be far too time-consuming and error prone. An approach for automated model extraction are proposed in [3] and a tool implementing this approach is in development, named *MXTC* – Model eXtraction Tool for C. The MXTC tool targets large implementations in C, consisting of millions of lines of code, and is based on a form of program slicing [23]. The model extraction tool was however not yet mature enough for producing real models for the HCRR evaluation in this paper. Due to the size of industrial systems, virtually all "dark corners" of the C programming language will be encountered, which leads to a quite complex tool, which must be very stable, and at the same time scalable to large quantities of code. However, an evaluation using MXTC and HCRR on a large industrial system is planned during 2009.

Problem Definition We can define the problem of best-effort response-time analysis with explicit input as follows. We are given a model of a real-time system, which can be simulated on simulation instances S, consisting of simulator parameters. Let $R(S)$ denote the highest response time measured for the task under analysis in the simulation of instance S. The goal of the problem is then to find a simulation instance S^* that maximizes R, subject to the constraints on S^* outlined in Section 8.2.1.

8.2.2 Input Representation

A simulation instance is a set of parameters that exactly determine the outcome of a simulation. In this paper, a simulation instance is represented using a set of sequences of integers, where each sequence is associated with either an arrival jitter of a task, an execution time, or an environmental input stimulus[1]. Each value then directly decides a selection of either jitter, execution time, or state in the task control flow. The advantage of this approach is that the direct relationship between representation and model properties makes it possible to locally refine specific aspects of a given simulation instance.

[1]Such environmental input stimulus is represented as various number of events generated by environmental task in Model 1 and Validation Model, and various execution time of Software Circuits (SWCs) in Model 2 in Section 8.4.

Let $\mathbf{J}_i$ be a sequence of actual jitter values j_{ir} experienced by instance r of a task τ_{ti}. We restrict j_{ir} to integer values in the interval $[0, \mathrm{ub}(\mathbf{J}_i)]$, where $\mathrm{ub}(\mathbf{J}_i)$ is an upper bound on jitter for task i in units of the smallest measurable time interval (clock ticks) for the target system. Furthermore, let $\mathbf{X}_k$ be a sequence of values for a certain environmental input stimulus or execution time in the simulated program, and X_k^j be the j^{th} such input value. We assume that all stimulus and execution times X_k^j are of integer type and have upper and lower bounds, so that $\mathrm{lb}(\mathbf{X}_k) \leq X_k^j \leq \mathrm{ub}(\mathbf{X}_k)$ for all k, j. Execution times are used only for deciding CPU time consumption of EXECUTE primitives. Bounds on execution times can be analysed using static analysis [24] or estimated through measurements.

A simulation instance S, defining a fully deterministic simulation of the model, is therefore a set

$$\{\mathbf{J}_1, \mathbf{J}_2, \ldots, \mathbf{J}_n, \mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_m\} \tag{8.1}$$

where n is the number of tasks which have non-zero jitter and m is the number of environmental stimulus and EXECUTE statements. Denote by N_i and M_k the number of values that are used to represent jitter sequence $\mathbf{J}_i$ and input sequence $\mathbf{X}_k$. N_i and M_k can be determined empirically by tracing how many values the simulator uses for each value. In theory, N_i and M_k can be unbounded, and for some long simulations, N_i and M_k may grow to unacceptable levels. In such cases, we suggest to set N_i and M_k to a fixed acceptable level. If there are not enough input values in the sequence, the simulator should report a warning, and start reuse values from the start of the sequence. For the evaluated models in this paper, N_i and M_k were long enough to represent all values used.

8.3 The Optimization Algorithm

In the rest of this paper, we focus on analysing the response time of a specific given task by varying the simulation instances used as input for the simulator. Analysis of an entire system can easily be done by performing our analysis several times, once for each task in the system.

8.3.1 Random Restart Hill Climbing

Our initial idea was to use a representation of the input parameters to RTSSim, which more directly corresponded to simulation parameters, in a full genetic algorithm [9]. However, initial experiments with the crossover operator, which

is the operator most often associated with genetic algorithms, proved unsuccessful and did not show any significant improvement over MABERA. Instead of focusing on the crossover operator, we chose to investigate iterative improvement of a single individual as an alternative. It turned out that hill-climbing [20], augmented with random restarts whenever a local minimum was detected, gave much better performance than MABERA.

The proposed new optimization algorithm, HCRR, is therefore based on hill climbing using random-restarts. Hill-climbing has the advantage of being one of the simplest metaheuristics available, and is based on the idea of starting at a random point, and then repeatedly taking small steps pointing upwards (wrt. the objective function, which in this paper is the measured response time) whenever such search directions exist. If no such steps exist, a local minimum may have been reached. Several techniques for escaping local minima exist (for example Tabu Search [8] and simulated annealing [12]), but a set of limited experiments conducted did not show any significant performance advantage over hill-climbing with random restarts.

Advantages of HCRR come from the combination of a strictly local improvement part, which quickly converges to high response times, with diversification mechanisms (jump-back to equal candidates, and full restarts) that are important to avoid local maxima. In contrast, MABERA doesn't employ such a mechanism, and consequently can easily get stuck in local optimas. In addition, the local improvement functionality of MABERA is inefficient in that it is not clearly connected to existing critical features of the solution candidate. Monte Carlo search, on the other hand, has no mechanism at all for local improvement, and therefore exhibits unsatisfactory convergence.

HCRR works by iteratively changing a small portion of the model parameter set, and restarts after a fixed number of non-improving simulations have been tried.

The implementation of HCRR is given in Algorithm 8.1. Here, the simulation budget is denoted $nofsims$, and $\mathrm{RT}(q)$ denotes the end time of the task under analysis in the simulation instance q when the worst response time occurred. The consumption time point of a simulation input X_i^j of any type (jitter, execution time, or environmental input stimulus) is expressed as TM_i^j. $q[X_i^j]$ is the current value of X_i^j in the simulation instance q. The function $\mathrm{rnd}(l, u)$ returns a random number between l and u if $l < u$; otherwise, it returns l. A completely random simulation instance can also be generated using the call $\mathrm{rnd\_inst}()$.

HCRR takes a currently best candidate ($best$) as input, which should be a random simulation instance when first called. It then begins by choosing as

Algorithm 8.1: Hill Climbing with Random Restarts

HCRR(*nofsims*,*m*,*k*,*best*)
  $curr \leftarrow$ MONTECARLO($\min(m, nofsims)$, rnd_inst())
  $nofsims \leftarrow nofsims - m$
  if $R(curr) > R(best)$ **then** $best \leftarrow curr$
  $\mathbf{E} \leftarrow \{curr\}$
  $nonimp \leftarrow 0$
  while $nofsims > 0$
    if $nonimp > nR$
      return HCRR(*nofsims*,*m*,*k*,*best*)
    else if $(nonimp + 1)$ mod $nB = nB$
      $curr \leftarrow$ random element in $\mathbf{E}$
    $nb \leftarrow$ NBH($curr$, $\lfloor k \cdot \mathrm{len}(curr) \rfloor$)
    SIMULATE(*nb*)
    $nofsims \leftarrow nofsims - 1$
    if $R(nb) > R(best)$ **then** $best \leftarrow nb$
    if $R(nb) > R(curr)$
      $curr \leftarrow nb$
      $\mathbf{E} \leftarrow \{nb\}$
      $nonimp \leftarrow 0$
    else
      $nonimp \leftarrow nonimp + 1$
      if $R(nb) = R(curr)$ **then** $\mathbf{E} \leftarrow \mathbf{E} \cup \{nb\}$
return *best*

starting point the best simulation instance from $\min(m, nofsims)$ randomly selected candidates using the MONTECARLO method. Then, in each iteration, $k \cdot \mathrm{len}(curr)$ random values of the current simulation instance $curr$ (which has $\mathrm{len}(curr)$ input values) used before $\mathrm{RT}(curr)$ are selected and modified using the neighborhood procedure NBH, shown in Algorithm 8.2.

The response time for the task under analysis is measured by running RTSSim using the SIMULATE(*nb*) call on a neighbor nb. Modifications suggested by NBH that increase response time are accepted, and changes that decrease response time are rejected. Modifications that have equal response time are rejected but saved for future reference, as described below.

A pure hill-climbing procedure is susceptible to getting stuck in local maxima, and can therefore exhibit less than satisfactory performance on many problems. In order to avoid convergence to locally optimal areas and to im-

Algorithm 8.2: Neighborhood procedure

NBH($inst, n$)
  for $k = 1$ **to** n
    $\mathbf{Q} = \{X_i^j \in inst \mid \mathrm{TM}_i^j < \mathrm{ET}(inst)\}$
    $X_i^j \leftarrow$ random input variable in $\mathbf{Q}$
    $V = \{\mathrm{lb}(\mathbf{X}_i) \ldots \mathrm{ub}(\mathbf{X}_i)\} \setminus \{inst[X_i^j]\}$
    $v \leftarrow$ random value in V
    $inst[X_i^j] \leftarrow v$

prove the probability of finding a true global maximum, two different diversification mechanisms were implemented. First of all, after nB non-improving iterations, the algorithm jumps back to a previously encountered, randomly selected simulation instance with an equal response time to the current instance. This distributes focus over a number of equal instances, which can help in avoiding small local maxima. The second technique is a common method for avoiding local maxima by restarting the hill-climbing procedure from a random location after a number of iterations. In HCRR, a random restart is performed after a sequence of nR non-improving iterations.

8.4 Case Studies

This section describes two industrial cases and one validation case in the form of simulation models. The models have similar architecture and analysis problems as two industrial real-time applications in use at ABB [1] and Arcticus Systems [4]. Although the simulation models contain relatively few tasks, at most 11, their behavioural complexity is significant due to e.g., shared variables, sporadic events and dynamic priority changes.

Model 1 (M1) is representing a control system for industrial robots developed by ABB Robotics, which is not possible to analyse using analytical methods such as RTA [5, 15]. This model has previously been used to evaluate MABERA in [14]. Model 2 (M2) is constructed from a test application used by Arcticus Systems [4], which develops the Rubus RTOS used in many vehicular systems. We also use a simplified version of Model 1 for validation (MV), where the code violating the assumptions of RTA has been removed. The purpose of this model is to investigate how close the response times found by HCRR are to the true worst-case response times derived by RTA.

The scheduling policy used is preemptive priority-based scheduling for all models. Models 2 and 3 use preemptive fixed-priority scheduling. Model 1

uses a preemptive scheduler and mainly static priorities, but contains one task that changes priority dynamically.

8.4.1 Model 1

This model describes a fictive system designed to be representative for a control system for industrial robotics, developed by ABB. The ABB system is quite large, containing around 3 millions lines of code and is not analysable using traditional analytical methods, such as RTA. Model 1 is of much smaller scale, but is designed to include some behavioural mechanisms from the ABB system which RTA can not take into account:

- tasks with intricate dependencies in temporal behaviour due to IPC and shared state variables;
- the use of buffered message queues for IPC, where triggering messages may be delayed;
- tasks that change scheduling priority or periods dynamically, in response to system events.

The modeled fictive system controls a set of electric motors based on periodic sensor readings and aperiodic events. The calculations necessary for a real control system are, however, not included in the model; the model only describes behaviour with a significant impact on the temporal behaviour of the system, such as resource usage (e.g., CPU time), task interactions and important state changes. The model contains four periodic tasks with the parameters shown in Table 8.1 (a lower valued priority is more significant).

Table 8.1: Task parameters for Model 1.

Task	*Priority*	*Period*	*Depends on*
PLAN	5	40000	UI
CTRL	4 or 2	10000 or 20000	PLAN, IO, UI
IO	3	5000	Sensor
DRIVE	1	2000	CRTL, UI

The environmental input stimulus in this problem is a sequence of integers from zero to two, denoting the number of external events that are generated by a sensor, measured in one IO task period. The IO task then sends equally many messages to the CTRL task. The CTRL task may change priority and

periodicity in response to two specific events in the model. The PLAN task is responsible for planning the movement of the physical object connected to the motors. The CTRL task calculates control signals for the motors with respect to coordinates sent from the PLAN task and IO events provided by the IO task. The DRIVE task actuates the motors based on the CTRL task output, which impact the execution time of the CTRL task.

The model also describes a user interface (UI) which generate sporadic events which impacts the system behaviour. There are three types of user interface events: START, STOP and GETSTATUS. The START and STOP events makes the system change between two system modes, IDLE and WORKING, with different temporal behaviours. The GETSTATUS event makes PLAN, CTRL and DRIVE send a status message to the user interface, which increases the execution time of those task instances. The task in focus of analysis is the CTRL task.

8.4.2 Model 2

This model describes a fictive system based on a test application from Arcticus systems, developers of the Rubus RTOS [4] which is used in heavy vehicles. This model uses a pipe-and-filter architecture, and contains 3 periodic transactions and one interrupt-driven task, in total 11 tasks. The inter-arrival time of the interrupt is 5000 simulation time units, with the offset and maximum jitter 500 and 100 simulation time units respectively. Tasks may trigger other tasks using trigger ports. The parameters of tasks and their execution times are given in Table 8.2.

This model is less complex than the two earlier models in that there exist no shared variables or IPC via message passing which can impact the tasks' timing and functional behaviour. Instead, the tasks have large variations in execution times, which makes the state space of this model very large. For this model, the evaluation focuses on the end-to-end response time of the transaction with a periodicity of 30,000 simulation time units, which also contains the tasks with the lowest priority.

8.4.3 Validation

Simulation-based methods for response-time analysis have in common that the result is not guaranteed to be a safe upper bound on the response time. We therefore constructed a validation model, analysable using RTA, with the purpose to investigate how close the response times given by HCRR are to the

Table 8.2: Task parameters for Model 2.

Task	*Period*	*Off.*	*Jitter*	*Prio.*	*Execution*
swcIT_1	5000	500	100	0	[100, 200]
swcIT_2	5000	500	100	0	[100, 200]
swcA_1	5000	0	0	1	[400, 500]
swcA_2	10000	0	0	1	[400, 500]
swcA_3	30000	0	0	1	[400, 500]
swcB_2	10000	0	0	1	[400, 500]
swcB_3	30000	0	0	1	[400, 500]
swcA_et2	10000	0	0	2	[500, 600]
swcA_et3	30000	0	0	2	[500, 600]
swcB_et2	10000	0	0	2	[500, 600]
swcC_et1	30000	0	0	2	[500, 600]

worst-case response times derived using RTA. Hence, RTA should provide an upper bound on the worst-case response time, which the simulation-based results should approach but not exceed. The validation model is based on Model 1, but with the following simplifications:

- Selected shared state variables are removed.
- Dynamic changes of priority and period are removed, only static attributes are used.
- Iteration loop bounds are added manually.

As a consequence, the validation model has considerably lower complexity, and exhibit quite different timing properties when compared to Model 1. For instance, the worst-case response time of the CTRL task (which as in Model 1 is the task under analysis) is only 52 % of the highest response times found for this task in Model 1.

Due to our extensive knowledge of this specific model, we could deduce that in order to improve the accuracy of the RTA (without being optimistic), the DRIVE task should be modeled as two separate tasks. These two tasks represent two different WCETs of the DRIVE task, depending on a rare sporadic event, where the minimum inter-arrival time is known. However, it is important to realize that such model refinements are hard to apply in practice, for real industrial systems, as the temporal behavior of such systems are rarely documented in detail. This refinement of the model had a major impact with

respect to RTA, yielding a worst-case response time of 4432 (refined model) instead of 5982 (without refinement).

8.5 Experimental Evaluation

This section presents an evaluation of accuracy, convergence and scaling properties of HCRR, using in total 7 different versions of the models described in Section 8.4. The experiments were done by running HCRR, a reimplementation of MABERA (MAB) and Monte Carlo (MC) simulation, on the three models previously described. Table 8.3 highlights the types of input parameters for the three models, i.e., the decision variables controlled by HCRR, MABERA or the Monte Carlo method.

Table 8.3: Simulator input parameters for the considered models.

Model	*Input Stimulus*	*Arrival Jitter*	*Execution Time*
Model 1	Variable	Variable	Constant
Model 2	N/A	0	Variable
Validation Model	Variable	Variable	Constant

The goal of the analysis is to find extreme response times for a specific task in the model. The results are, with the exception of Figure 8.1, obtained from running 100 samples of each algorithm and test case, each sample being allowed to run 10,000 simulations, in order to get a good comparison for a fixed time length. The simulation budget was considered reasonable due to the convergence of HCRR on our most realistic model (Model 1). The experiments were performed on an Intel Core 2 Duo, 2.33 GHz with 2 GB of RAM.

For MABERA, the population was obtained by scaling the population size of 10,000 used in [14] to reflect the change in number of simulations per sample. The ratio is 81,400 in [14] to 10,000 in this paper. As a result, we use a population size of 1250, which is 1/8 of the original population size. The same fraction of parents as for the original method is used, which translates to a selection of 12 parents in each generation. For each of these, 104 mutations are generated. In order to ensure that MABERA used exactly 10,000 simulations in total, the original termination threshold was disabled.

For the parameters in HCRR, the jump-back threshold (nB) should be relatively small to spread the search over the set of equal candidate solutions found so far. However, the random restart threshold (nR) should be larger in order not to erase any progress made so far, but small enough to force restart from a

local minimum as soon as possible. The fraction k of input values changed in each iteration should provide a good balance between power (larger fractions) and low dimensionality (smaller fractions).

To select the parameters for HCRR, we performed a small number of sequential experiments on Model 1, varying one parameter at a time. For each parameter set, we measured the convergence as the average best result in any iteration (i.e., simulation) for 20 sample runs, or more formally:

$$C = \frac{\sum_{i=1}^{20} \sum_{j=1}^{S} R_i^j}{20 \cdot S}$$

where S is the number of simulations and R_i^j denotes the response time found after j simulations in sample run i. The number of simulations was 500 for nB and k and 3000 for nR. The parameters giving quickest convergence ($nB = 2$, $nR = 300$, and $k = 0.02$) were then used for all experiments. The results of the experiments are shown in Table 8.4.

Table 8.4: Parameter selection.

$nB = nR = \infty$		$k = 0.02, nR = \infty$		$k = 0.02, nB = 2$	
k	C	nB	C	nR	C
0.01	7796.76	100	7931.37	1000	8308.11
0.02	**8010.90**	50	7902.86	300	**8312.05**
0.03	7988.83	20	7939.70	100	8304.17
0.04	7976.14	10	7972.72	50	8254.26
0.05	7961.80	7	7992.25		
0.07	7944.69	5	7944.27		
0.10	7761.59	4	8001.89		
0.15	7645.62	3	7919.24		
0.20	7604.48	2	**8024.98**		
0.30	7483.33	1	7944.27		

To show the effects of scaling on the three algorithms, Model 1 is used to create larger systems by instantiating several independent instances of it, thereby creating independent "subsystems" where each subsystem is a complete model as described in Section 8.4, including tasks, input events, state variables and message queues. The subsystems are completely independent, except that they share the same CPU. The model setup can be described using the following parameters:

SUBSYSTEMS: The number of subsystems to use, varied between 1 and 4.

CPU_SPEED: The scale factor for all execution times. Let C be the original execution time for a single EXECUTE statement in the model, then $C/\mathrm{CPU\_SPEED}$ is the resulting execution time in the multiply instantiated model.

OFFSET: The relative offsets between subsystems, allowing for different "phasings" between subsystems. Throughout the experiments, a phasing of 20000 time units has been used.

To avoid priority clashes, new priorities are assigned using the formula $P^n = 10P^o + I$, where P^n is the new priority, P^o is the old priority, and I is the subsystem index. For CPU_SPEED we use factors of 1.0, 1.5, 1.8 and 2.2 when having 1, 2, 3 and 4 subsystem instances respectively.

8.5.1 Timing Results

The obtained lower bounds on worst-case response time are illustrated by the following labels:

MC: The traditional Monte Carlo approach to generate simulation instances using random input data.

MAB: The MABERA approach, using a population size of 1250 of which 12 parents are selected for reproduction, unless stated otherwise. The algorithm is modified to run for a limited number of simulations.

HCRR: The new algorithm based on random restart hill climbing. The algorithm is given in Algorithm 8.1.

Figure 8.1 shows the results obtained for Model 1 from Section 8.4.1. The top of the figure contains the response time distributions of the three algorithms, where the MABERA results are taken from [14]. Results were obtained using 200 sample runs for MABERA, 200 runs for MC, and 100 runs for HCRR. For MABERA and MC, each sample required on average 81,400 simulations. Each HCRR sample was allowed 10,000 simulations. The bottom of Figure 8.1 shows convergence (mean RT and 95 % confidence intervals), using the standard parameters of 10,000 simulations, for the three algorithms with 100 samples for each algorithm.

The upper part of Figure 8.1 shows that HCRR managed to find the highest known response time, 8474, in all 100 sample runs. The highest response time found by MABERA was 8349, and this value was only found one single time. The MC approach managed to find a maximum response time of 8390, which is also found once. Note that HCRR was only allowed approximately 12 % of the

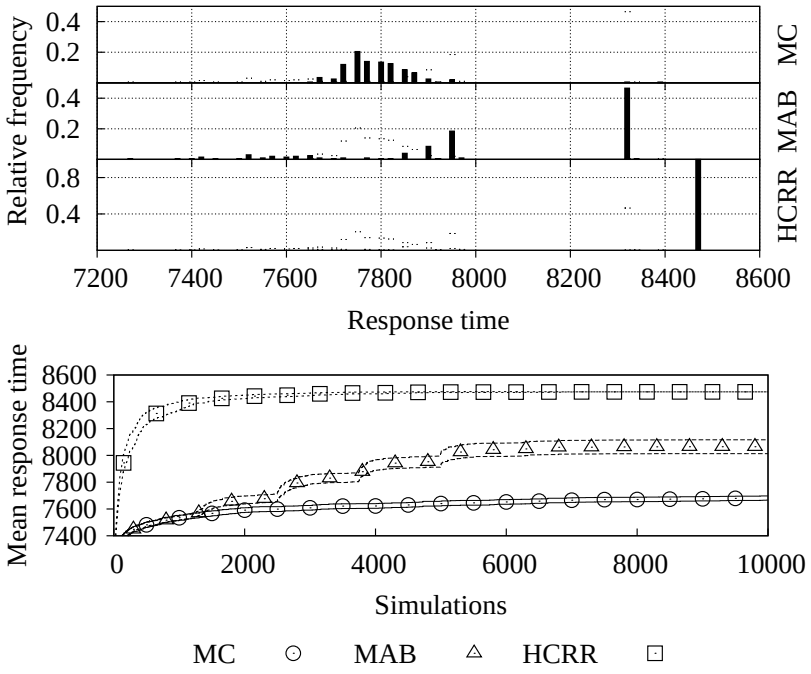


Figure 8.1: Final RT distributions and convergence (mean RT and 95 % confidence intervals) for model 1.

number of simulations used by MC and MABERA. If we compare the number of simulations done when the highest known response time was found, HCRR was approximately 1628 times faster than MABERA and MC. The runtimes for one sample of all algorithms were less than 3 minutes.

Figure 8.2 shows the obtained results for Model 2 (Section 8.4.2) using the standard parameters. In this model, the tasks have large variations in execution times, which makes the state space very large. We can see that HCRR yields a result approximately 5 % higher than what is obtained from the two other methods. Interestingly, it looks like HCRR was still slowly progressing towards higher response times at 10,000 simulations, while both MABERA and MC seems to have converged quite early to a much lower result. For Model 2, all algorithms finished in less than one minute per sample.

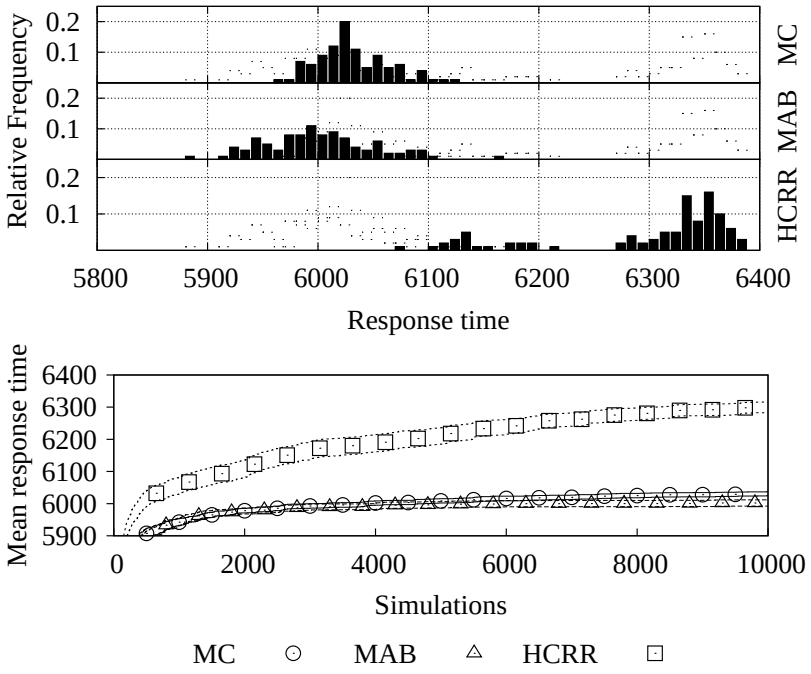


Figure 8.2: Final RT distributions and convergence (mean RT and 95 % confidence intervals) for model 2.

In Figure 8.3, we can see the results for the validation model described in Section 8.4.3, again using the standard parameters. In addition, we show the RTA results. Here, HCRR could find a response time of 4432 in every sample run, which was also confirmed by RTA to be the worst-case response time. As before, the difference between MABERA and MC appears to be quite small. MABERA found the worst case in a few samples, while MC did not, but it is questionable if the difference is statistically significant. For the validation model, MC took less than 50 seconds, MABERA less than 130 seconds, and HCRR less than 90 seconds for one sample run.

Figure 8.4 shows how the different methods scale to larger systems, by illustrating the convergence for Model 1 when increasing the model size to 2, 3 and 4 subsystems (model instances). As expected, since the state space increases with number of subsystems, all three algorithms converge slower when

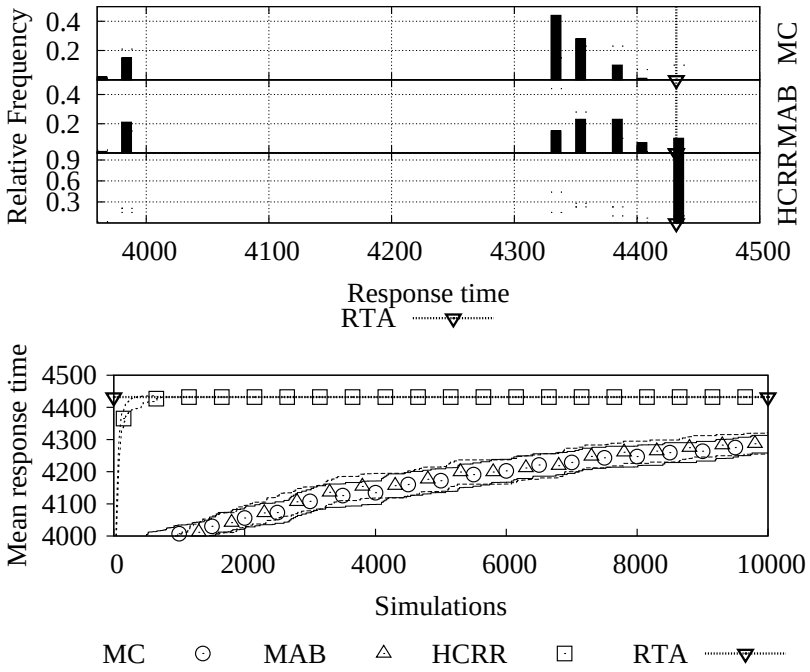


Figure 8.3: Final RT distributions and convergence (mean RT and 95 % confidence intervals) for the validation model.

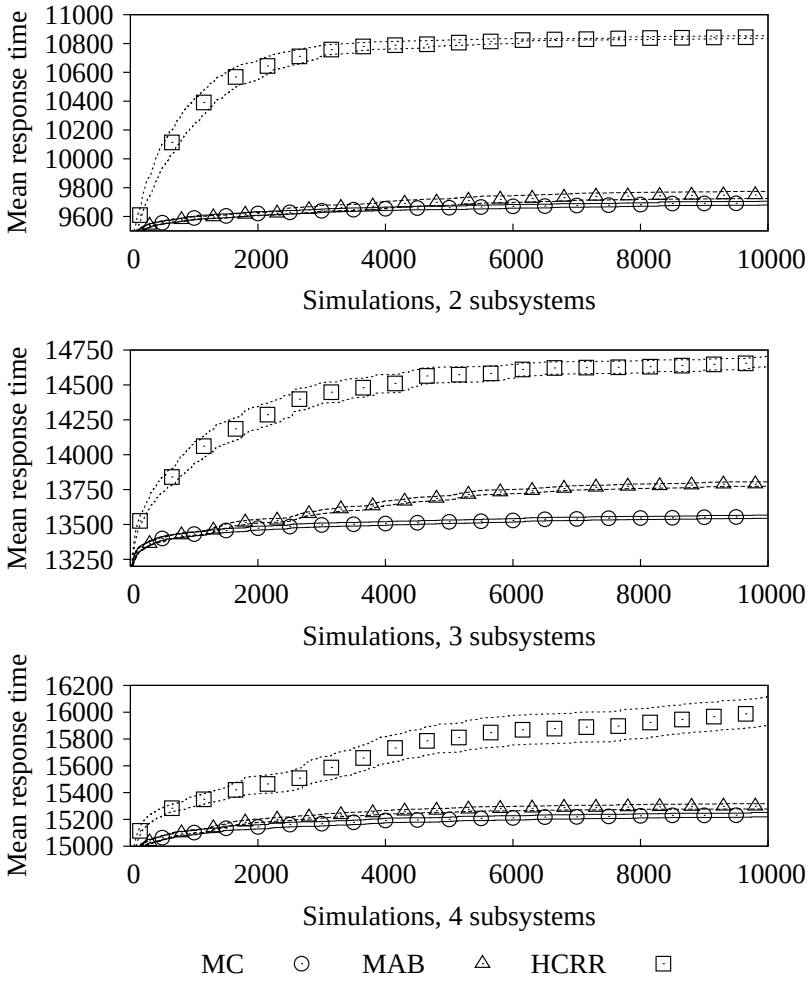


Figure 8.4: Convergence (mean RT and 95 % confidence intervals) for model 1 using 2-4 subsystems.

system size is increased. For two subsystems, HCRR is consistently better than both MC and MABERA, with all results reported being higher than the maximum result found for both MC and MABERA. The results for 3 and 4 subsystems indicate that the difference between the methods decrease as system size is increased, although HCRR produced on average 4.7 to 11 % higher results than both MC and MABERA. For 4 subsystems, neither of the methods appear to have converged. However, during the 10,000 simulations, HCRR progressed more quickly to higher response times than both MC and MABERA. Runtimes for a single sample when having 2 subsystems were below 4, 7 and 5 minutes for MC, MABERA and HCRR. Sample runtimes were below 5, 10 and 6 minutes for 3 subsystems and below 8, 16 and 10 minutes for 4 subsystems.

Table 8.5: Average end result and point when HCRR passes the second best end result.

	MC	*MABERA*	*HCRR*	*Passes 2^{nd} best*
M1-1	7682	8065	8474	224
M1-2	9693	9750	10844	238
M1-3	13555	13789	14672	521
M1-4	15235	15298	16013	764
M2	6031	6002	6299	634
MV	4286	4288	4432	89

The average end results are summarized in Table 8.5. The last column also shows the average number of simulations needed for HCRR to obtain the end result of the second best method (using 10,000 simulations). As we can see, HCRR reached the second-best result 13 to 112 times faster than the second-best method did. For all tried models, HCRR on average outperformed the other methods in less than 800 simulations, which corresponds to less than 1.5 minutes of computation time on the PC used for the experiments.

8.5.2 Average Convergence

To measure average convergence more exactly, we use the relative difference in average response-time results over a time span of d simulations. We say that a method has for practical purposes converged (on average) when

$$1 - \overline{R}^{(k-d)} / \overline{R}^{(k)} \quad \leq \quad \varepsilon \tag{8.2}$$

where $\overline{R}^{(k)}$ is the average response-time result at simulation k for a set of samples. Using this definition, convergence will never be detected before at least d simulations has been performed. In order to measure convergence for the evaluation presented in this paper, d obviously needs to be less than the number of simulations (10,000) performed in each sample. We therefore use $d = 1000$ for the convergence comparison. For the tolerance parameter, we chose a value of $\varepsilon = 0.001$. In other words, if the average progress in 1000 simulations is lower than 0.1%, we declare that the method has converged on average. It should be pointed out that different parameters will give radically different results on convergence, and true convergence is reached and detected only when $\varepsilon = 0$ and d is sufficiently large.

Table 8.6: Convergence on iteration k to response time $\overline{R}^{(k)}$ for the different methods.

	MC		*MABERA*		*HCRR*	
	k	$\overline{R}^{(k)}$	k	$\overline{R}^{(k)}$	k	$\overline{R}^{(k)}$
M1-1	7632	7670	7356	8062	4090	8466
M1-2	4806	9660	6518	9728	7093	10830
M1-3	3527	13502	7801	13773	5568	14578
M1-4	3410	15175	5104	15271	6948	15881
M2	3656	5997	3552	5991	9556	6295
MV	–	–	–	–	1661	4432

Table 8.6 summarizes the convergence results, obtained from Equation. (8.2) with the parameters above, for Model 1 with 1-4 subsystems (M1-1 to M1-4), Model 2 (M2), and the validation model (MV). In general, we can see that HCRR converged to significantly higher response times than MABERA and MC. For the validation model, the only method to converge within 10,000 simulations was HCRR. Overall, the results are mostly consistent with what can be seen in Figure 8.1, 8.2 and 8.3, but also classified the slow average progress for HCRR on M2 in Figure 8.2 as convergence. Running the algorithm longer would either yield slightly higher results or confirm convergence.

For M1-4, convergence of HCRR is also detected in iteration 6948 after a slow progress between simulation 6000 and 8000, but as we can see in Figure 8.4, more average progress is made after simulation 8000. Sampling more than 100 runs for M1-4 would most likely even out the slope after simulation 6000. In any case, HCRR has clearly not converged after 10,000 simulations, and running the algorithm longer would likely yield even higher results.

8.6 Conclusions

Simulation-based analysis of complex real-time systems has the potential to provide engineers with timing properties of real-time systems not conforming to classical real-time analysis models such as Response-Time Analysis (RTA). In this paper, a new best-effort approach for simulation-based timing analysis has been presented, and the new algorithm, based on Hill Climbing with Random Restarts (HCRR), is shown in our evaluation to find more accurate worst-case response time faster than alternative methods such as MABERA and Monte Carlo simulation.

In evaluating HCRR, three models of industrial real-time systems have been simulated, and the results show that HCRR was 4-11 % more accurate than the second-best method, and between 13 to 112 times quicker in reaching the end result of the second-best method. In one case, HCRR was 1628 times quicker in finding its more accurate result than the second-best method. An analysis of convergence indicate that for two cases out of six, even higher respones times could be achieved by letting HCRR run longer.

Industrial deployment of HCRR requires an efficient method for extracting simulation models from complex software systems. A tool for that purpose, MXTC, is currently in development. This uses mainly static analysis, but also measurements in order to obtain execution-time data for the model. The simulation model analyzed by HCRR could however use data from WCET analysis tools as well, for supported hardware platforms. The execution-time measurements requires context-switch recording with accurate timestamps. This is possible in most real-time operating systems.

Acknowledgement

This work was supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS. We are grateful to Jan Carlson, Mikael Sjödin, and Björn Lisper for comments and improvement suggestions.

Bibliography

[1] *ABB Group*. Web page, http://www.abb.com.

[2] J. ALANDER, T. MANTERE, G. MOGHADAMPOUR, AND J. MATILA, *Searching Protection Relay Response Time Extremes Using Genetic Algorithm — Software Quality by Optimization*, in Proceedings of the In-

ternational Conference on Advances in Power System Control, Operation and Management, vol. 1, 1997, pp. 95–99.

[3] J. ANDERSSON, J. HUSELIUS, C. NORSTRÖM, AND A. WALL, *Extracting Simulation Models from Complex Embedded Real-Time Systems*, in Proceedings of the International Conference on Software Engineering Advances, ICSEA'06, IEEE, Oct. 2006.

[4] *Arcticus Systems*. Web page, http://www.arcticus-systems.com.

[5] N. AUDSLEY, A. BURNS, R. DAVIS, K. TINDELL, AND A. WELLINGS, *Fixed Priority Pre-emptive Scheduling: an Historical Perspective*, Real-Time Systems, 8 (1995), pp. 129–154.

[6] G. BEHRMANN, A. DAVID, J. HÅKANSSON, M. HENDRIKS, K. G. LARSEN, P. PETTERSSON, AND W. YI, *UPPAAL 4.0*, in Proceedings of the International Conference on Quantitative Evaluation of Systems, 2006.

[7] D. DECOTIGNY AND I. PUAUT, *ARTISST: An Extensible and Modular Simulation Tool for Real-Time Systems*, in Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.

[8] F. GLOVER AND M. LAGUNA, *Tabu Search*, in Modern Heuristic Techniques for Combinatorial Optimization, C. R. Reeves, ed., McGraw-Hill, 1995, ch. 3, pp. 70–150.

[9] D. E. GOLDBERG, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Professional, January 1989.

[10] M. JOSEPH AND P. PANDYA, *Finding Response Times in a Real-Time System*, The Computer Journal, 29 (1986), pp. 390–395.

[11] K. KIM, J. L. DIAZ, L. L. BELLO, J. M. LOPEZ, C.-G. LEE, AND S. L. MIN, *An Exact Stochastic Analysis of Priority-Driven Periodic Real-Time Systems and Its Approximations*, IEEE Transactions on Computers, 54 (2005), pp. 1460–1466.

[12] S. KIRKPATRICK, C. D. GELATT, AND M. P. VECCHI, *Optimization by Simulated Annealing*, Science, 220 (1983), pp. 671–680.

[13] J. KRAFT, *RTSSim – A Simulation Framework for Complex Embedded Systems*, Tech. Report., Mälardalen University, March 2009.

[14] J. KRAFT, Y. LU, C. NORSTRÖM, AND A. WALL, *A Metaheuristic Approach for Best Effort Timing Analysis Targeting Complex Legacy Real-Time Systems*, in Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2008.

[15] C. LIU AND J. LAYLAND, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, 20 (1973), pp. 46–61.

[16] F. MUELLER AND J. WEGENER, *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints*, Real-Time Systems, 21 (2001), pp. 268–241.

[17] R. NOSSAL AND T. M. GALLA, *Solving NP-Complete Problems in Real-Time System Design by Multichromosome Genetic Algorithms*, in Proceedings of the SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems, 1997, pp. 68–76.

[18] R. RACU AND R. ERNST, *Scheduling Anomaly Detection and Optimisation for Distributed Systems with Preemptive Task-Sets*, in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, April 2006, pp. 325–334.

[19] *Rapita Systems*. Web page, http://www.rapitasystems.com, 2008.

[20] S. RUSSELL AND P. NORVIG, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Second ed., 2003.

[21] S. SAMII, S. RAFILIU, P. ELES, AND Z. PENG, *A Simulation Methodology for Worst-Case Response Time Estimation of Distributed Real-Time Systems*, in Proceedings of Design, Automation and Test in Europe, vol. 10-14, IEEE, Mar. 2008, pp. 556–561.

[22] *UPPAAL Website*. Web page, http://www.uppaal.com, 2008.

[23] M. WEISER, *Program Slicing*, in Proceedings of the International Conference on Software Engineering, IEEE Press, 1981, pp. 439–449.

[24] R. WILHELM, J. ENGBLOM, A. ERMEDAHL, N. HOLSTI, S. THESING, D. WHALLEY, G. BERNAT, C. FERDINAND, R. HECKMANN, T. MITRA, F. MUELLER, I. PUAUT, P. PUSCHNER, J. STASCHULAT, AND P. STENSTRÖM, *The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools*, ACM Transactions in Embedded Computing Systems, 7 (2008), pp. 1–53.

Chapter 9

Paper D: Reducing Vehicle Maintenance using Condition Monitoring and Dynamic Planning

Markus Bohlin, Malin Forsgren, Anders Holst, Björn Levin, Martin Aronsson and Rebecca Steinert.

In Proceedings of the 4th IET International Conference on Railway Condition Monitoring.

18–20 June 2008, Derby, United Kingdom of Great Britain and Northern Ireland.

Abstract

It is a common fear in industry that introducing condition based maintenance (CBM), with its constant monitoring of several subsystems, will lead to more frequent service interventions compared to traditional cyclic maintenance, effectively countering the potential value of implementing CBM. Because of this, adoption of CBM must be done with great care, and the maintenance organization and planning process needs to be geared for more flexibility. To harvest the potential value in CBM for rail vehicles, we propose to combine condition monitoring with online maintenance planning. We use an adaptive planning software module to quickly find new suitable vehicle movement plans, and a heuristic packing module to reconstruct maintenance packages with as few maintenance stops as possible. This prevents vehicles from visiting the maintenance depot too frequently. At the same time, we actively keep the risk of breakdowns low. Evaluation of our methods in a simulated environment for train operation and maintenance, using real-world time tables and vehicle plans, show that by taking the operation times of individual components into account, it is indeed possible to reduce the amount of maintenance as well as the number of service interventions significantly compared to traditional cyclic maintenance.

9.1 Background

Advanced methods for effective maintenance planning have been under development for several decades [4, 8, 14]. Still, a common maintenance approach within production, manufacturing, industry, transportation etc., is cyclic maintenance planning, where maintenance is carried out within predetermined intervals without taking the equipment condition into account [6, 17–19].

The period between service interventions is often based on expert knowledge about the equipment usage, its lifetime and the rate of deterioration [9, 16, 20, 23]. The main reason for performing cyclic maintenance is that maintenance costs and equipment availability are easier accounted for in the general production plan and budget. A consequence, however, is that parts of the equipment may be replaced much earlier than their optimal replacement time [6]. Another consequence is the increased risk of unplanned maintenance (caused by unexpected failures), which can be far more costly than planned service interventions [20]. Both cases show that cyclic maintenance can cause unnecessary maintenance expenses [18].

In recent years, the interest in developing and improving methods that are not based on cyclic maintenance has increased. Such approaches are for example condition based maintenance (CBM) and predictive maintenance (PdM) [2, 11, 12]. These approaches are in general based on the desire to reduce maintenance costs by performing maintenance on equipment only when necessary [3, 5]. For this purpose, various metrics (such as distance travelled, hours of operation, the number of times a door has been opened and closed, etc.), and sensor-based methods (such as oil analysis, vibration analysis, etc.) can be used to continuously measure and monitor the condition of the equipment [10, 13].

Prediction models for when maintenance should be performed on equipment introduce more uncertainty and complexity into the planning process, compared to cyclic maintenance. Also, there is a fear in industry that the number of service interventions will increase using condition based approaches, since the maintenance planning is based on the independent condition of individual subsystems rather than on the system as a whole as in the cyclic maintenance approach.

In other words, since each subsystem reaches its replacement time independently, there is a risk of much more frequent, small interventions instead of a few, large, pre-scheduled ones. Therefore, it is desirable to combine subsystems that are in need of service, immediately or in the near future, into service packages that are fitted into pre-planned maintenance opportunities. This approach is usually referred to as opportunistic maintenance (OM) [15, 19, 21, 22].

The idea of OM is to combine equipment parts in need of service that are functionally or economically dependent, such that the overall maintenance cost is reduced. Opportunistic maintenance is for the same reasons also desirable with condition based approaches.

We propose a method that takes maintenance scheduling a step further than OM. Instead of performing maintenance on service packages at pre-planned and fixed opportunities, service packages are maintained at dynamically planned maintenance opportunities based on the condition of the whole system as well as equipment recommendations provided by the manufacturers. The main goal is the same as with OM – lowering maintenance costs – without introducing unnecessary complexity into the production planning process. The potential gain of condition based opportunistic maintenance is even greater than for regular OM, since the observed increase in lifetime of components under condition monitoring can be utilized to a fuller extent.

The focus in this paper is on lowering maintenance costs by reducing the number of maintenance stops and interventions. The method is based on an idea of more efficiently using subsystem information, through a better model for handling uncertainty in use and wear. This is used to improve the ability of an optimizer to repeatedly or continuously improve the maintenance schedule.

9.2 Contribution

In this paper, we present a dynamic approach for combining prediction of maintenance deadlines with repeated re-planning of vehicle use and maintenance actions. Our aim is to develop a maintenance scheme that is simple and causes minimal change to established practices, while giving large savings in maintenance costs. We also use subsystem information (e.g. subsystem counters) more efficiently, by better modelling the way in which certainty in the prediction of the actual maintenance deadline increases with time.

Our approach is based on frequent re-optimization of the production and maintenance plans for each unit, in this paper manifested by a train in a fleet. For each candidate solution investigated, the service packages needed are constructed and adapted to the maintenance need of the situation and the candidate solution. The plans are re-optimized as soon as significant changes in the planned operation or in the system condition are detected, taking into account information of average wear and variance, predicted wear from planned unusual loads, planned use, availability requirements, etc. The most common change that triggers a re-optimization of the maintenance plan, is the continuous reduction in uncertainty between the operational counters of individual

subsystems (e.g. the number of hours on the compressor or the number of cycles of the doors) and the global counter for the whole system (in this case the aggregated distance travelled by the train) as the system is being used (see Section 3 below). We also assume that each subsystem has a predefined operational limit based on recommendations from the manufacturer. One could introduce a more complex model, taking into account the actual probability density of failures given the use of each component. However, our assumption, and the way we re-optimize the maintenance plan, has the advantage of not increasing the risk of breakdowns in any way, since the limits used are identical to the ones used to plan the traditional cyclic maintenance schedule.

We have tested our method in a simulator for train traffic, developed in an earlier project, using real time tables and realistic fault frequency data as input, in addition to recommended service intervals for different systems (see Section 6). From the simulation results we show that our method indeed reduces the number of service interventions to well below the level for cyclic maintenance while reducing maintenance costs and without increasing the risk of failures. With relatively simple measures and available service data, our method is also relatively easily applicable in practice.

9.3 Wear model

We assume that the unit to maintain consists of a number of components or subsystems. The manufacturer provides a recommended maximum operation interval before service for each component, measured in some unit (e.g. hours, kilometres, or cycles). We also assume that service of components was originally lumped together into predefined service packages, each with its own global recommended maximum operation length. The latter is presumably chosen to keep the risk of any of the included components running over its recommended length below some low level.

We wish to compare the situation when service is performed based on this global operation length with when service is based on the operation lengths of individual components, with respect to both the amount of maintenance and the number of service interventions. Specifically, we want to see if the number of service interventions can be reduced in spite of basing service on the individual components' operation lengths.

There are two different sources of uncertainty that should not be confused with each other. The first concerns at which operation length a failure will occur for a specific component. This uncertainty is not the focus of this study, but we will assume that if service is performed before the recommended operation

length for each component, then the risk of failure will remain below some acceptable level, identical to that of the cyclic maintenance approach.

The other uncertainty concerns the relation between the individual operation lengths and the global operation length, i.e. how fast the usage counters on the subsystems advance as the global usage counter increases. A typical example is the maintenance of trains, where maintenance is based primarily on the travelled distance of the vehicle, whereas the use of the onboard equipment to a large extent is governed by the characteristics of the specific routes traversed by the vehicle, or by other elements that are unrelated to the kilometres travelled. For instance, the number of times the doors will be opened per kilometre naturally depends on how many stops the train makes, which in turn depends on the time table. Similarly, the HVAC (air conditioner) unit will be used more on a warm day than on a cold day. Since many of these conditions are unknown when the traditional cyclic maintenance schedule is determined, a large safety margin must be included when selecting the maximum global operation length. It is the reduction of this uncertainty, between the global operation length and that of the components, that we wish to harness.

When designing a traditional cyclic maintenance schedule based on a global operation counter, an assumption has to be made concerning the rate at which each subsystem counter is advancing relative to the global counter. In our case, this is equivalent to determining a distribution over how far the global counter has advanced before the subsystem counter has reached its limit. To ensure that the risk of running a subsystem counter over its limit is lower than some acceptable level, the limit for this subsystem on the global counter must be chosen a certain amount earlier than the expected average value (see Figure 9.1). Differently put, a fairly large safety margin is needed if the usage varies a lot.

To be more precise, let us assume that the increase, c, of a subsystem counter is approximately normally distributed, with parameters that depend on the increase, d, of the global counter:

$$c \sim N(k_m d, k_s \sqrt{d}) \tag{9.1}$$

where k_m and k_s are constants. To keep the risk sufficiently low, the service deadline for the global counter, d_0, must be selected such that there is a number, a, of standard deviations left before the subsystem service deadline, c_0, is reached:

$$c_0 = k_m d_0 + a k_s \sqrt{d_0} \tag{9.2}$$

In the simulations the level of uncertainty is a parameter which is varied,

whereas the global deadline d_0 is fixed to make the results correspond to a fixed risk level. The uncertainty is then measured as the factor, f, between the deadline d_0 and the global counter increase, d_m, that would give the subsystem deadline c_0 on average (i.e. without the margin of a standard deviations):

$$c_o = k_m d_m = k_m d_0 f \tag{9.3}$$

This gives the standard deviation constant k_s as:

$$k_s = \frac{(f-1)\sqrt{d_o}}{a} k_m \tag{9.4}$$

Given the above equations (9.1), (9.2), (9.3) and (9.4), we can generate a random increase in each subsystem counter c given an increase in the global counter d, and then calculate a new global deadline d_0 from the remaining subsystem lifetimes c_0.

The reduction of uncertainty that we benefit from, stems from the fact that, at each point where we re-calculate the maintenance schedule, there is no uncertainty in the events that have already passed. We know, e.g., exactly how many times the doors were opened for the distance that has been travelled. We only have an uncertainty and a safety margin for the distance that will be travelled in the future.

Note that if there is no uncertainty, then there is nothing to gain with dynamic planning since we have assumed that the recommended operation limit is a hard deadline: We need to get to the depot with a frequency determined by the most constraining limit. Also, if one limit is much tighter than the others (and the uncertainty is less than this difference), then there are again no alternatives: The global limit should be selected as the most constraining limit and counted in the same unit.

The interesting case is when there are several components with limits of the same order as the most constraining one, and they have a random variation relative to each other. In the simulations (see Section 6 below), we have therefore used ten components with the same mean and variance relative to the global limit. This seems a realistic assumption for many systems. Also note that although the components in the simulations had the same mean and variance, they were sampled independently and consequently diverged in their behaviour in the same way as components in real systems do and in the same way that has been feared to cause the many extra service interventions.

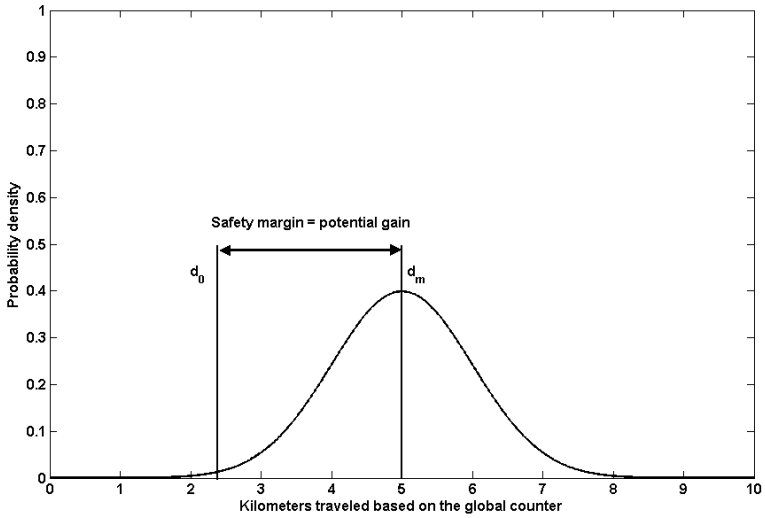


Figure 9.1: The probability density for the increase in a global counter until a subsystem counter reaches its deadline. The global deadline d_0 must be well before the expected counter increase d_m, unless our method is used.

9.4 Construction of service packages

When maintenance deadlines are not fixed but variable in terms of calendar time during operation, service needs to be organized into packages dynamically in order to reap the benefits of CBM.

We call each time period in which maintenance can be done a maintenance *opportunity*. Each opportunity i starts at time point t_i and has a fixed capacity c_i, which we assume is the length of the time period for the opportunity. The goal of the service packing problem is then, for a fixed plan of operation, to allocate individual maintenance activities j, each with a deadline at e_j and duration of d_j time units, to the opportunities given by the maintenance plan, such that no deadline is missed. If we use $a(j)$ to denote the opportunity that j is allocated to, we get the constraint of equation (9.5).

$$\forall j : t_{a(j)} \leq e_j \tag{9.5}$$

In addition, the total duration of the maintenance activities allocated to an opportunity should not exceed its capacity, or more formally expressed by equation (9.6):

$$\forall i : \sum_{j:a(j)=i} d_j \leq c_i \tag{9.6}$$

The activities allocated to a single opportunity form a *service package*.

In preventive maintenance optimization, a common assumption often made is that the machinery to be maintained will run for an infinite amount of time, and that preventive maintenance is periodic. The objective of optimization then becomes finding the optimal recurring maintenance intervals for the components of the system.

If maintenance opportunities are unlimited in duration (that is, for all opportunities i it holds that $c_i = \infty$) and the single objective is to reduce the average number of depot stops per time unit, then, under these assumptions, there exists a simple maintenance packing strategy which is optimal: Define the activity with the earliest deadline as the *critical* activity, and schedule service for this activity at the latest possible time before the deadline. For obvious reasons, the maintenance opportunity chosen this way for the critical activity is inevitable. Once we have such a depot stop, we create a service package that includes all activities that do not depend on a preceding activity, in addition to the critical activity. Clearly, no other strategy can generate fewer service packages per time unit.

In reality, these conditions rarely hold. For example, maintenance is usually planned for depot stops of limited durations only, and there are often several optimization criteria, such as unused lifetime of components, cost of downtime at different calendar dates, and risk factors associated with doing maintenance too late, that must be taken into account and balanced against each other. Therefore, we use a slightly more advanced service packing algorithm, which can be more easily tailored for different objectives.

In order to reduce the number of non-empty service packages, we employ a simple allocation strategy that works as follows. Maintenance activities for a single vehicle are sorted according to their deadlines, with the activity that needs service at the earliest point in time first. Then, in turn, we try to find a suitable opportunity for each activity. The search for a suitable maintenance opportunity is done in two phases. In the first phase, we only consider opportunities which are already non-empty. The possible such non-empty opportunities that do not occur too early or too late are tried in turn according to date, starting with the latest one first. The first opportunity found whose capacity can

also hold the activity in question is selected for assignment. In the case that no such opportunity is found, we enter the second phase where all opportunities are tried, including the empty ones. We once again search for opportunities in reverse chronological order, and the first one that can hold the activity in question is selected for assignment.

In addition, we can also control the length of the time interval in which we search for suitable opportunities as assignment candidates. We do this by varying a parameter p and search only for opportunities i for which equation (9.7) hold.

$$pe_j \leq t_i \leq e_j \tag{9.7}$$

In our experiments, we let $p = 0.5$.

9.5 Routing of vehicles

In order to meet maintenance requirements and to reduce the number of maintenance stops to a minimum, vehicles must be routed dynamically to and from the depot. We formulate the routing problem as follows.

We are given a set of transports T and a set of vehicles V. Each vehicle i has a set of deadlines $E_i = \{e_{i1}, e_{i2}, ..., e_{il}\}$ for component/subsystem maintenance, identical to the deadlines in Section 4. Each vehicle also has an initial transport I_i. Each transport has a departure and arrival station, departure and arrival time, and a length in kilometres.

A feasible vehicle plan for a vehicle is a cyclic sequence of transports, starting with the initial transport, where consecutive transports arrive to and depart from the same station, respecting departure and arrival times. All transports should be allocated to exactly one vehicle. In addition, there are constraints on maintenance that need to be fulfilled. Each vehicle should be maintained in time, that is, all deadlines should be preceded by a corresponding maintenance activity.

The problem of determining optimal vehicle routes is NP-hard in general [7], and therefore we chose to use a heuristic method to find routes that are suitable. In the simulator, component wear is simulated for each transport in order of arrival time, and for each simulation step, a number of candidate plans are generated and evaluated using a cost function. Each candidate plan consists of a vehicle plan and a maintenance plan, generated as described in Section 4, and the selection of candidate plans is based on the manual process used today for vehicle re-planning. The method we use for vehicle routing is

simpler than, for example, the one found in [1], but serves the purpose of aiding service package re-planning well enough.

Two vehicles that are stationary at the same time on a given station can, given enough spare time, swap future vehicle plans. In essence, our method is to apply a small set of such swaps between pairs of vehicles in order to improve the current situation. In practice, we generate a set of modified plans and evaluate the set using the cost function described below, selecting the plan with the lowest cost for execution. For all vehicles i which miss a deadline, we generate and evaluate all possible swaps before the deadline miss between i and other vehicles that occupy the same station at the same time.

The evaluation is done as follows. For each vehicle plan, the allocation algorithm in Section 4 is applied, resulting in a new maintenance plan. Each plan consists of assigned maintenance dates m_{ij} for each vehicle i and active maintenance deadline e_{ij}. Define $r_{ij} = e_i - m_i$ as the lifetime remaining when service is carried out.

Ideally, we want r_{ij} to be positive and small, but not smaller than an extra safety margin s. We penalize maintenance with w_E per kilometre for early maintenance and w_L per kilometre for late maintenance. In addition, we use a large base cost B_M for deadline misses and a cost B_O per used maintenance opportunity. Formally, we have the following cost function per vehicle i and maintenance deadline j:

$$C_{ij} = \begin{cases} w_E(r_{ij} - s) & \text{if } r_{ij} > s \\ w_L(s - r_{ij}) & \text{if } 0 \leq r_{ij} \leq s \\ B_M + w_L s + w_M r_{ij} & \text{if } r_{ij} < 0 \end{cases} \tag{9.8}$$

Denote the number of used opportunities as x. The total cost for all vehicles then becomes

$$C = \sum i \sum j C_{ij} + B_O x \tag{9.9}$$

In our experiments, we used as s the original deadline divided by 12, and let $w_E = 0.1$, $w_L = 3.0$, $B_M = 10000$ and $B_O = 100$.

9.6 Test case

We have evaluated our method using a simulator of train movements and repair shop activity that was developed in a previous project. The simulator uses actual service data from Bombardier Regina trains operated in the Mälardalen region in Sweden, and accumulates wear as described under “Wear model”

above. We have during these tests focused on reducing the number of service interventions, to make the point of this paper clear. In a more practical setting, other objective functions could be used instead, such as the total maintenance cost.

For the simulator setup, we used actual time tables and vehicle plans for 10 vehicles from the autumn of 2003 as a basis for our experiments. To be able to evaluate our method, we created 10 artificial components for each vehicle, each taking 6 minutes to maintain and having a base deadline of 7200 km of operation. Re-planning of maintenance was done as soon as any vehicle arrived at a station. To reduce planning time to an acceptable level, we used a planning horizon of 30 days for the experiments. The planning horizon is stable because the vehicle plans include several stops at the depot within this time period, and the maintenance deadline used is usually covered within a week of typical train operation. Maintenance opportunities were assumed to be present whenever trains visited the single maintenance depot used for the particular time period and train operator.

The pre-existing time tables and vehicle plans already accounted for vehicle maintenance in that vehicles passed the depot approximately once a week. To add flexibility and maintenance opportunities, we augmented the vehicle plans with two extra vehicles stationed in the depot. For these vehicles we also added pre-scheduled trips to and from the depot to a larger hub nearby. These trips could be used on demand when trains needed to be rerouted to the depot, which is in practice necessary to obtain flexibility and ability to schedule maintenance at a precise date.

9.7 Results

As mentioned in the Background section, there is a risk that savings gained from condition based maintenance are lost in repeated visits to the maintenance shop. In Figure 9.2, we observe that by using our method, the number of service interventions is in fact reduced to well below the level of traditional, cyclic maintenance. This is achieved while retaining a substantial part of the cost reduction for the actual maintenance actions and without increasing the risk of breakdowns. Conclusively, our proposed approach can reduce maintenance costs without introducing any major negative side effects.

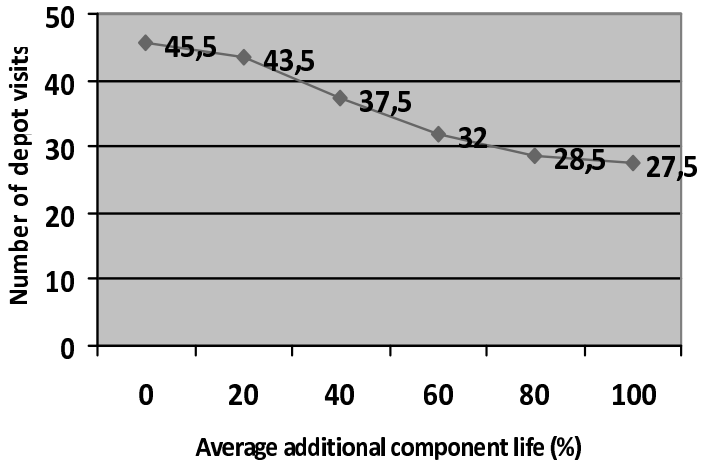


Figure 9.2: The reduction in number of visits to the maintenance shop as a function of actual additional component lifetime after the replacement that would occur using traditional cyclic maintenance.

9.8 Discussion

In this paper, we propose an approach at maintenance planning for rail vehicles based on the monitoring of individual subsystem counters, such as compressor run time and number of door open-close cycles. Our approach is easy to implement, since subsystem counters are already available in many cases. The key to harvesting the gains of condition monitoring is to repeatedly re-construct maintenance packages based on wear estimates. In this way, we can reduce the need for some of the large safety margins usually present in most maintenance schemes without increasing the risk of failure.

A natural next step is to build a wear model that better handles the other source of uncertainty, i.e. the lifetime of a component or subsystem given its own usage counter. This can either be done based on the statistics available to e.g. an operator of a fleet, or it can be based on manufacturer supplied statistics. However, in both cases it is non-trivial to ensure that the selected risk level is not exceeded in the system.

The maintenance packing algorithm used today is based on heuristics and is therefore in many cases not optimal. Our planned future work includes investigating better optimization models and algorithms, capturing more of the maintenance planning process such as resource requirements, cost of spare parts and manpower, availability, life-cycle costs, etc. Yet another extension would be to take care of repairs, handling spare part needs using error codes and preliminary root-cause analysis. Such extensions would most likely require more exchange of information between the actors involved.

Acknowledgement

This work has been funded by VINNOVA, The Swedish Governmental Agency for Innovation Systems.

Bibliography

[1] L. ANDEREGG, S. EIDENBENZ, M. GANTENBEIN, C. STAMM, D. S. TAYLOR, B. WEBER, AND P. WIDMAYER, *Train Routing Algorithms: Concepts, Design Choices, and Practical*, in Proceedings of the 5th Workshop on Algorithm Engineering and Experiments, Society for Industrial and Applied Mathematics, 2003, pp. 106–118.

[2] M. BEVILACQUA AND M. BRAGLIA, *The Analytic Hierarchy Process Applied to Maintenance Strategy Selection*, Reliability Engineering & System Safety, 70 (2000), pp. 71–83.

[3] A. CHELBI AND D. AIT-KADI, *Inspection and Predictive Maintenance Strategies*, International Journal of Computer Integrated Manufacturing, 11 (1998), pp. 226–231.

[4] R. DEKKER, *Applications of Maintenance Optimization Models: A Review and Analysis*, Reliability Engineering & System Safety, 51 (1996), pp. 229–240.

[5] R. DEKKER AND P. SCARF, *On the Impact of Optimisation Models in Maintenance Decision Making: the State of the Art*, Reliability Engineering & System Safety, 60 (1998), pp. 111–119.

[6] J. ENDRENYI, S. ABORESHEID, R. ALLAN, G. ANDERS, S. ASGARPOOR, R. BILLINTON, N. CHOWDHURY, E. DIALYNAS, M. FIPPER,

R. FLETCHER, C. GRIGG, J. MCCALLEY, S. MELIOPOULOS, T. MIELNIK, P. NITU, N. RAU, N. REPPEN, L. SALVADERI, A. SCHNEIDER, AND C. SINGH, *The Present Status of Maintenance Strategies and the Impact of Maintenance on Reliability*, IEEE Transactions on Power Systems, 16 (2001), pp. 638–646.

[7] T. ERLEBACH, M. GANTENBEIN, D. HÜRLIMANN, G. NEYER, A. PAGOURTZIS, P. PENNA, K. SCHLUDE, K. STEINHÖFEL, D. S. TAYLOR, AND P. WIDMAYER, *On the Complexity of Train Assignment Problems*, in Proceedings of the 12th International Symposium on Algorithms and Computation, London, UK, 2001, Springer-Verlag, pp. 390–402.

[8] A. GARG AND S. DESHMUKH, *Maintenance Management: Literature Review and Directions*, Journal of Quality in Maintenance Engineering, 12 (2006), pp. 205–238.

[9] A. JARDINE, D. BANJEVIC, AND V. MAKIS, *Optimal replacement policy and the structure of software for condition-based maintenance*, Journal of Quality in Maintenance Engineering, 3 (1997), pp. 109–119.

[10] A. JARDINE, D. BANJEVIC, M. WISEMAN, S. BUCK, AND T. JOSEPH, *Optimizing a Mine Haul Truck Wheel Motors Condition Monitoring Program Use of Proportional Hazards Modeling*, Journal of Quality in Maintenance Engineering, 7 (2001), pp. 286–302.

[11] H.-B. JUN, D. KIRITSIS, M. GAMBERA, AND P. XIROUCHAKIS, *Predictive Algorithm to Determine The Suitable Time to Change Automotive Engine Oil*, Computers & Industrial Engineering, 51 (2006), pp. 671–683.

[12] L. MAILLART AND S. POLLOCK, *Cost-optimal condition-monitoring for predictive maintenance of 2-phase systems*, IEEE Transactions on Reliability, 51 (2002), pp. 322–330.

[13] L. MANN, A. SAXENA, AND G. M. KNAPP, *Statistical-Based or Condition-Based Preventive Maintenance?*, Journal of Quality in Maintenance Engineering, 1 (1995), pp. 46–59.

[14] J. J. MCCALL, *Maintenance Policies for Stochastically Failing Equipment: A Survey*, Management Science, 11 (1965), pp. 493–524.

[15] A. NEELAKANTESWARA RAO AND B. BHADURY, *Opportunistic maintenance of multi-equipment system: a case study*, Quality and Reliability Engineering International, 16 (2000), pp. 487–500.

[16] P. A. SCARF, *On the Application of Mathematical Models in Maintenance*, European Journal of Operations Research, 99 (1997), pp. 493–506.

[17] A. TSANG, *Strategic dimensions of maintenance management*, Journal of Quality in Maintenance Engineering, 8 (12 April 2002), pp. 7–39.

[18] J. VELASQUEZ, R. VILLAFAFILA, P. LLORET, L. MOLAS, A. SUMPER, S. GALCERAN, AND A. SUDRIA, *Development and Implementation of a Condition Monitoring System in a Substation*, in Proceedings of the 9th International Conference on Electrical Power Quality and Utilisation, October 2007, pp. 1–5.

[19] H. WANG, *A Survey of Maintenance Policies of Deteriorating Systems*, European Journal of Operations Research, 139 (2002), pp. 469–489.

[20] R. C. M. YAM, P. TSE, L. LI, AND P. TU, *Intelligent predictive decision support system for condition-based maintenance*, The International Journal of Advanced Manufacturing Technology, 17 (2001), pp. 383–391.

[21] X. ZHENG AND N. FARD, *A Maintenance Policy for Repairable Systems Based on Opportunistic Failure-Rate Tolerance*, IEEE Transactions on Reliability, 40 (1991), pp. 237–244.

[22] X. ZHOU, L. XI, AND J. LEE, *A dynamic opportunistic maintenance policy for continuously monitored systems*, Journal of Quality in Maintenance Engineering, 12 (2006), pp. 294–305.

[23] X. ZHOU, L. XI, AND J. LEE, *Reliability-centered predictive maintenance scheduling for a continuously monitored system subject to degradation*, Reliability Engineering & System Safety, 92 (2007), pp. 530–534.

Chapter 10

Paper E: Optimization of condition-based maintenance for industrial gas turbines: Requirements and results

Markus Bohlin, Mathias Wärja, Anders Holst, Pontus Slottner and Kivanc Doganay.

Paper number GT2009-59935.

In Proceedings of ASME Turbo Expo 2009: Power for Land, Sea and Air.

June 8–12, 2009, Orlando, Florida, USA.

Abstract

In oil and gas applications, the careful planning and execution of preventive maintenance is important due to the high costs associated with shutdown of critical equipment. Optimization and lifetime management for equipment such as gas turbines is therefore crucial in order to achieve high availability and reliability. In this paper, a novel condition-based gas turbine maintenance strategy is described and evaluated. Using custom-made gas turbine maintenance planning software, maintenance is repeatedly reoptimized to fit into the time intervals where production losses are least costly and result in the lowest possible impact. The strategy focuses on accurate online lifetime estimates for gas turbine components, where algorithms predicting future maintenance requirements are used to produce maintenance deadlines. This ensures that the gas turbines are maintained in accordance with the conditions on site. To show the feasibility and economic effects of a customer-adapted maintenance planning process, the maintenance plan for a gas turbine used in a real-world scenario is optimized using a combinatorial optimization algorithm and input from gas turbine operation data, maintenance schedules and operator requirements. The approach was validated through the inspection of a reference gas turbine after a predetermined time interval. It is shown that savings may be substantial compared to a traditional preventive maintenance plan. In the evaluation, typical cost reductions range from 25 to 65 %. The calculated availability increase in practice is estimated to range from 0.5 to 1 %. In addition, downtime reductions of approximately 12 % are expected, due solely to improved planning. This indicates significant improvements.

NOMENCLATURE

CBM	Condition-based Maintenance
EOC	Equivalent Operating Cycles
EOH	Equivalent Operating Hours
INSP	Inspection
OEM	Original Equipment Manufacturer
PM-opt	Preventive Maintenance Optimizer
REPL	Replacement
RFC	Retirement For Cause

10.1 Introduction

Condition-based Maintenance (CBM) is a term for methods and methodology that, based on the actual condition and predicted future use, in theory allows maintenance to be performed at the best possible date for each component. Typical applications include components that do not fail instantaneously, but deteriorate in a quantifiable and, preferably, observable way over a period of time. An early failure indication enables the user to avoid the consequences of an unexpected breakdown. Early signs can be detected by the use of diagnostic equipment and/or by analytical calculations taking the actual service conditions of the equipment into account — so-called prognostics. According to [9], equipment operators increasingly use condition-based maintenance instead of, or in addition to, scheduled maintenance to reduce lifetime equipment operating costs.

However, merely having diagnostics and/or prognostics is not enough to derive all or even most benefits from CBM. In [10], it is stated that in order to maximize the benefits from CBM for the enterprise, it is as important to focus on the aftermarket supply chain — i.e. the back-end of the process, including maintenance — as it is to develop better data gathering, diagnostic and prognostic techniques. Further, it is shown that optimizing the value chain results in lower costs and higher availability.

In practice, better knowledge of the actual maintenance requirements of the components of a machine should be reflected in maintenance intervals dynamically adapted to the current condition and predicted usage of the components. For gas turbines, predictions regarding future component condition and lifetime are based on factors such as load profile, quality of fuel, ambient temperature, particle levels, and so on. To maximize the benefits of CBM, maintenance also needs to be replanned whenever the current condition and future

predictions, and hence the future maintenance intervals, change significantly.

With a growing emphasis on life cycle cost reduction for capital equipment such as gas turbines, equipment operators are increasingly investigating potential cost reductions. One way to minimize life cycle costs and maximize earnings is to optimize maintenance according to a customer's specific condition. Achieving an optimal or near-optimal maintenance plan, which minimizes the total cost, depends on the availability of diagnostics and prognostics, as well as on maintenance planning technologies. Successful planning also involves developing accurate and comprehensive user knowledge, in part because solutions engineered for one user can then be adapted to the specific needs of other users [2].

The maintenance process adopted in this paper combines condition information with the requirements of the operator. This is done in order to carry out maintenance as efficiently as possible, thus ensuring that potential short-term profits will be evaluated in an overriding life cycle cost perspective. To manage all relevant information, a preventive maintenance optimization tool (PM-opt) has been developed. PM-opt plans preventive maintenance for complex technical systems and maximizes earnings for a system operator. This is done through the use of an advanced prognosis process and input from an operator regarding operation profile, ambient conditions and financial data such as production value and standstill costs. This information is processed in PM-opt, generating an optimized preventive maintenance schedule adapted to an operation-unique situation, hence maximizing profit. The process is also supported by an advanced diagnostic tool to further increase reliability and availability.

The goal is to provide operating conditions that will increase availability with predictable scheduled maintenance, based on condition-monitoring assessment, with little or no downtime during deployments. Any changes in, for example, operation profile will instantly affect the preventive maintenance. Also, if an unplanned opportunity occurs, maintenance can be re-scheduled if it is profitable to use this 'new slot'. PM-opt can deal with these situations and re-optimize maintenance if this is financially justifiable for the operator of the gas turbine.

10.2 Gas Turbine Maintenance

For gas turbines, maintenance planning is usually done many months in advance due to the user's cost associated with a period of gas turbine inactivity. The date and duration of a maintenance period are determined in advance to coincide where possible with other scheduled stops, such as plant shutdowns

and vacations. CBM offers a potentially more flexible approach. In it, the amount of flexibility depends on factors such as risk willingness, condition of components, value of production and future operation profile. The major advantages of CBM include the possibility to adapt the maintenance plan according to user-specific demands and needs and transparency regarding the possible consequences of different choices. Also, each component can be utilized as efficiently as possible. This reduces downtime and costs and increases the potential earnings due to an increased number of operating hours.

10.2.1 Equivalent Operating Hours and Cycles

Today, maintenance schedules are often based on EOH/EOC calculations, which model the equivalent operating hours and cycles used under different operational conditions. In a time interval from 0 to T, EOH is calculated as

$$\mathrm{EOH} = 5 \times \mathrm{EOC} + \int_{t=0}^{T} C_x(t) C_f(t) C_w(t) C_{T\gamma diff}(t)\, dt \tag{10.1}$$

where EOC is the total number of cycles. The factors $C_x(t), C_f(t), C_w(t)$ and $C_{T\gamma diff}(t)$ are used to model operational conditions; $C_x(t)$ depends on load, $C_f(t)$ on fuel quality, $C_w(t)$ on the presence of water injection, and $C_{T\gamma diff}$ on the presence of a significant exhaust temperature difference. However, the model is rather coarse in how these variables are handled. Further, factors such as ambient air temperature and pressure, rotational speeds, and outlet temperatures are not taken into sufficient consideration. Instead, the EOH calculations have substantial built-in safety margins to accommodate for variations and conditions not explicitly modeled. A more detailed model is therefore used for the CBM approach in this paper.

10.2.2 Condition-based Maintenance

The benefits of utilizing CBM to reduce lifecycle cost compared to a time-based preventive approach have been well-documented over the years. However, CBM has become a catch-all term for any type of health monitoring [7]. A majority of papers written on CBM or Health Management of Machines focuses on the areas of diagnostics and prognostics. There are many techniques and emerging technologies that are making engine monitoring more complete and informative. Basic parameter monitoring can and does provide valuable information on the performance of an engine [3]. Through intelligent processing and integration with other parameters, valuable information can be acquired,

including actual life consumed, life remaining, and the condition of the gas turbine relating to its operation profile and ambient conditions.

These approaches mean dealing with a large amount of data. For this reason, statistical approaches are becoming popular tools for life management [13]. Applications include calculating the risk involved in extending the life of components. For example, Retirement For Cause (RFC) is an approach that allows each component in an engine to be used for the full extent of its safe life [5]. Statistical approaches like Weibull analysis are popular in the world of industrial gas turbines. Original equipment manufacturers (OEM's) usually have databases on the number of parts retired from service as a function of operating parameters. These samples can, however, be quite small, and the root cause may not be properly identified.

10.2.3 Component Life Assessment

Maintenance is all about increasing equipment life by activities that verify component functionality and detect faults before they occur. A typical scenario is seen in Fig. 10.1. The component life is assessed at each maintenance event and, depending on the result, new optimistic and pessimistic curves are drawn [12].

A gas turbine component is subjected to a number of potential damage mechanisms, related to the temperature and load that it is exposed to, the specific environment it operates in, or a combination thereof. In essence, the lifetime is estimated using models of gas turbine component deterioration that, given engine state parameters like rotational speeds, ambient air pressure and temperature, compressor outlet temperature, estimated turbine inlet temperature and pressure, turbine exhaust temperature and an increment length, return the approximate life consumption during the analyzed time increment. A similar approach is used to predict fatigue damage. However, instead of a time increment length, a number of cycle parameters like loading rate and dwell time within certain load ranges would be used. The models were developed using available knowledge from calculations and field experience. This is described in more detail in [12, 15].

A prognostics tool uses the deterioration models to calculate the residual lifetime (depending on a customer's operation profile, environmental conditions and actual gas turbine data). It can be described as an advanced EOH/EOC calculator, keeping track of every damage location on the gas turbine's components. When a certain damage location reaches a predetermined limit, either an inspection or a replacement is necessary. (see Fig. 10.2). As the operation

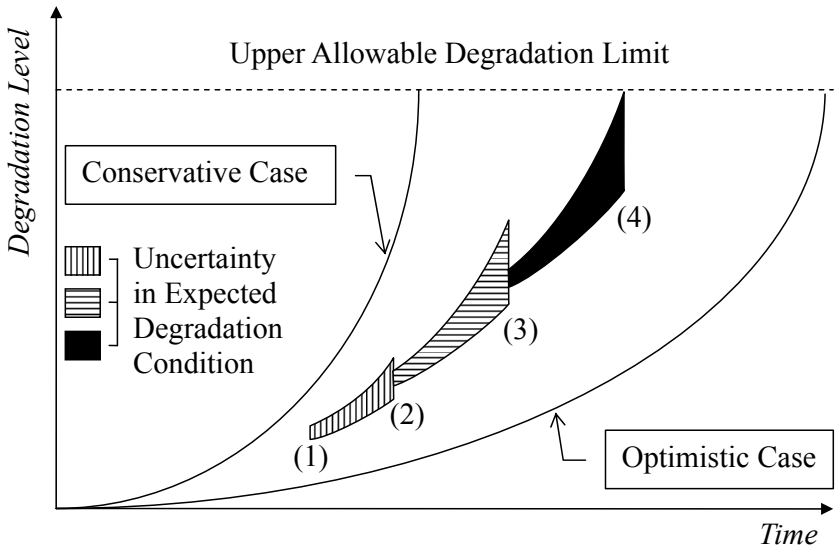


Figure 10.1: Component maintenance activities and life extension.

commences, EOH/EOC is accumulated on each damage location according to the deterioration models. The location with the highest amount of accumulated damage represents the condition of the component. The reaction time is the time needed for an OEM to plan and execute a maintenance task; this varies with different components.

The operator's specific characteristics are used, in combination with knowledge about the wear of components, to customize an optimal or near-optimal maintenance plan for the gas turbine components. The maintenance plan is also dynamically updated in order to increase the precision in the prediction of the point in time at which a planned maintenance action must be performed. Figure 10.3 describes the dynamic maintenance plan and how information is gathered during the operation to more precisely estimate the point in time for a maintenance action. Here, an inspection done at time x is premature in the sense that not enough observable damage has been accumulated. Therefore,

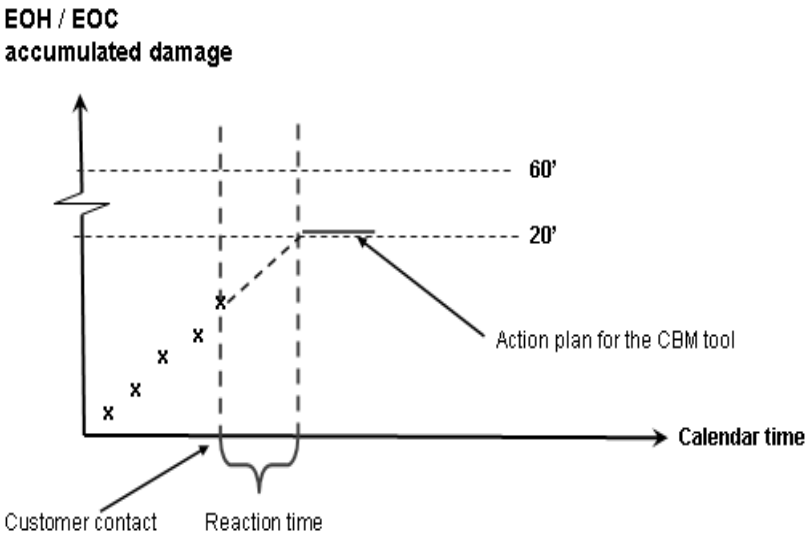


Figure 10.2: EOH/EOC accumulator.

inspecting the turbine at a later time y will decrease the uncertainty of when replacement of the worn component has to be done. Also, the risk associated with postponing an action can be calculated.

10.2.4 Component Life Extension and Risks

Every life extension beyond the normal operation regime means that there is a risk of triggering a damage mechanism that has not previously been considered for the component in question (due to initial design requirements or due to the lack of understanding and knowledge of the damage mechanism in question). Damage mechanisms may also interact in ways previously unheard of. In theory, therefore, every increase in life time means a risk increase that needs to be addressed.

If the life extension is based upon a prognostics analysis, where life times are based upon actual service conditions rather than design point conditions, the risk of failure due to any of the typical design damage mechanisms like creep, oxidation and fatigue, should be taken into account. The remaining risks

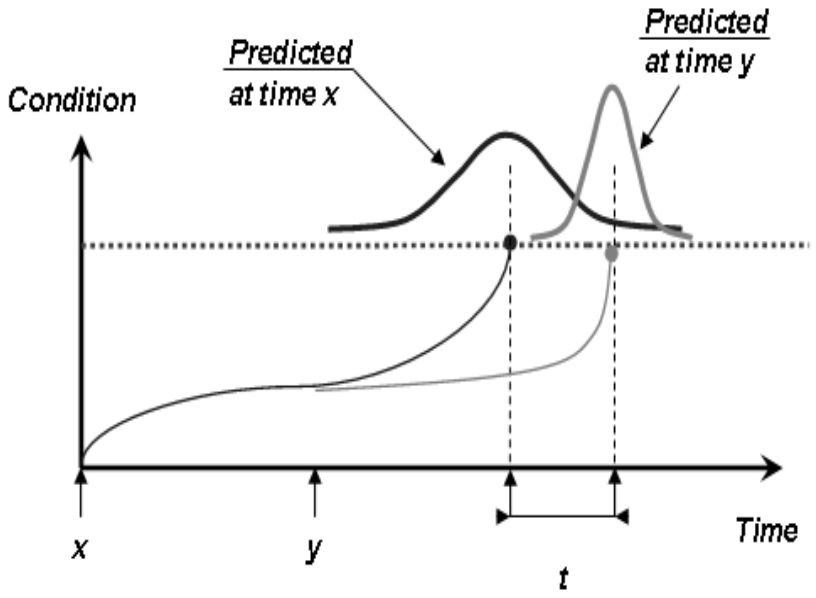


Figure 10.3: Point in time for planned maintenance action.

are due to previously unseen failure mechanisms, and due to new combinations of damage mechanisms including material deterioration. Both of these are real and have to be seriously considered. Due to the huge potential in extending service life, it is in many cases desirable to do so anyway. In order to do this with minimum risk, it is proposed the extension be accomplished in the following way:

- Considering type of material, operation temperatures and load, determine which known damage mechanisms and combinations thereof could possibly occur in each relevant component.
- Based upon what can possibly occur, define component inspections that can detect upcoming damage with sufficient accuracy.
- Based upon how good the inspection methods are, that is, how far in advance they can be expected to detect upcoming damage, determine suitable maximum inspection intervals.

When life is prolonged, performing the risk mitigation inspections described above at the defined maximum inspection intervals is strongly recommended, even if the prognostics model shows a life increase of several magnitudes.

As a gas turbine accumulates operation hours it will be possible to see some patterns during inspections and overhauls. By extracting sample components during inspections, the status of one set of components, as compared to standard lifetimes and to other sets of components operating under the same conditions, can be determined. This means that the benefits of the procedure indicated in Fig. 10.1 are used. However, usually the gas turbine is disassembled at least once before most components are scrapped, allowing access to the components for detailed status assessment. Therefore, by picking sample components for destructive testing even before reaching their expected end of life, valuable knowledge about actual service conditions can be gained and service life can be extended accordingly. This also means that even moderate lifetime improvements become much more valuable to the end user [12].

If the components in one stage “always” look notably good, this should mean that there is something in the conditions at the site that is lenient to the components. This also means that when adding a new set of components, the expectation on their actual life could be increased somewhat and the “halfway” inspection can be adjusted towards the half-life that should be expected in this specific gas turbine.

Postponing the first inspection beyond the “safe life” limit of the component should not be recommended, however, since there is always a possibility that conditions may have changed to the worse. Additional safety margins may be desired depending on the circumstances.

10.3 Gas Turbine Maintenance Planning

One of the major questions, if not the most important one, to be answered by persons responsible for the maintenance management of gas turbines, is when to do a maintenance action: “The maintenance action is due at a certain date — can it be postponed?” This optimization problem clearly belongs to the advanced maintenance sphere. This and other questions regarding optimal or near-optimal decision making is becoming more and more important together with an increased focus on cost minimization and profitability within the gas turbine sector. The type of problem dealt with is however of a very complex nature, and the question is therefore difficult to answer simply. To handle the complexity of the problem, one option is to use combinatorial optimization

methods, which can handle a wide variety of side constraints and different costs.

10.3.1 Maintenance Optimization

Because frequent preventive maintenance is costly, optimal or near-optimal maintenance planning is a primary interest for many gas turbine users. A well-known fact is that components in a gas turbine face different wear, depending on parameters such as environment, load, events, fuel type, and the like. This means that two identical gas turbines with different operators can present significant differences in wear. In order to make an optimization, every component in a gas turbine must be monitored and the accumulated equivalent operating hours (EOH) and equivalent operating cycles (EOC) must be considered. In addition, predictions based on an estimate of the expected future wear should be available in order to compute expected maintenance deadlines.

The wear of components of equal type can be expected to be somewhat similar. Therefore, several components are usually grouped into a set, represented by a single (composed) component. As an example, a single component could be used for all guide vanes in the first stage of the gas turbine. The residual lifetime of the composed component is represented by the most worn component in this set. As long as operation is normal and a component is not subjected to damage (e.g., due to foreign objects or deviation in temperature profile causing uneven wear), all components in a set are replaced at the same time. However, if components face unequal wear, the replacement of such components must be done with great care. Failure to do this properly may result in sub-optimization, causing increased costs for an operator due to higher maintenance frequency as a function of uneven component wear. This is avoided by the use of a highly detailed component database with component traceability, ensuring an effective replacement schedule throughout the gas turbine's lifetime.

10.3.2 Maintenance Setup

Considering only the cost of spare parts, a theoretically optimal maintenance strategy is to always perform maintenance as late as possible. The condition, and hence the deadlines, of the components in the gas turbine change stochastically throughout the lifetime of the turbine due to environmental factors such as particle levels, fuel quality, load, temperature, moisture levels, to name a few. This will also spread out the individual maintenance deadlines over time. The theoretically optimal strategy will thus result in many maintenance stops spread

out more or less evenly, in turn resulting in a maintenance schedule unacceptable to many customers. Thus, a strategy for the co-allocation of maintenance in order to reduce the number of maintenance stops is required. A flexible strategy that can meet a wide variety of maintenance requirements is to use optimization. There, a cost function for the maintenance schedule is minimized while a set of requirements on the resulting plan are satisfied.

Components in PM-opt

In PM-opt, each component is associated with 1) a set of maintenance activities with specified duration and cost, and 2) a maintenance schedule, which is a sequence of activities (replacements and inspections) that should be repeated according to a specified pattern. The component also handles its current condition, which is measured in the consumed lifetime of the component schedule, thus indicating, according to the best knowledge at that time instant, the exact "location" in the component maintenance schedule.

Maintenance Items

In order to model maintenance to a sufficient level of detail, a single gas turbine component is represented in PM-opt by a sequence of maintenance items. A maintenance item is the basic activity the optimizer handles, and corresponds to the individual, more or less regular, maintenance activities done at a maintenance stop. Items should be allocated to maintenance stops, and thus constitute one of the major parameters in the time to solve the problem.

Maintenance items are divided into two different main types: inspections and replacements of a component. Reconditioning a component is modeled using a replacement maintenance item. Regardless of its main type, each maintenance item has a maximum uptime before that maintenance item has to be done (typically the maintenance interval length before an inspection or a replacement) and a maintenance type, which contains data on costs and durations associated with the exact type of maintenance. The origin time point from which this maximum uptime is measured is different depending on type.

Each maintenance item also has data regarding the actual maintenance activities carried out (for example, work time consumption and a cost specification for the maintenance). In addition to this, maintenance items can be restricted to occur in a certain calendar interval, defined by an earliest and a latest date for maintenance. Items with due dates typically depend on the dates of other items. A set of maintenance activities associated with a component can be visualized as a tree of dependencies regarding maximum uptime, as shown

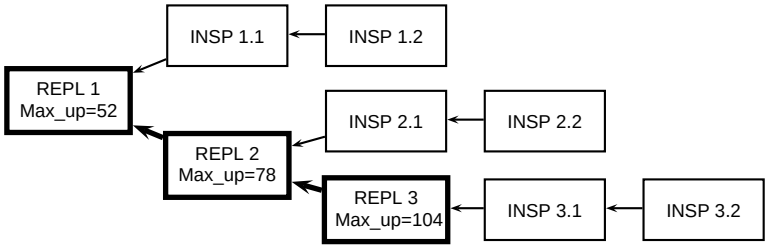


Figure 10.4: Replacement type items and their dependencies.

in Fig. 10.4 and 10.5.

Replacement Maintenance Items

For replacement items, uptime is measured from the preceding replacement maintenance item for the same component. The outcome of an inspection could potentially affect the lifetime predictions of a component. However, since this is unknown at planning time, this information has to be fed into the optimizer after the inspection has been done and when the results are available.

In addition to a maximum uptime, each replacement maintenance item also has a minimum uptime that has to pass before a replacement is even allowed for this component, as well as an earliest and latest calendar date for maintenance. Figure 10.4 shows an example of replacements and their time dependencies. Since replacements (REPL in Fig. 10.4 and 10.5) can also represent overhauls and other significant maintenance activities, it is possible to differ between such activities and have different inspection schedules for different replacement items.

Inspection Maintenance Items

For inspection maintenance items, uptime is measured from the previous inspection of the same component. Uptime is measured from the preceding replacement item if there is no preceding inspection item. As shown in Fig. 10.5, each replacement is followed by a number of inspections of the component in question. When a component is replaced, a new inspection schedule is rolled out.

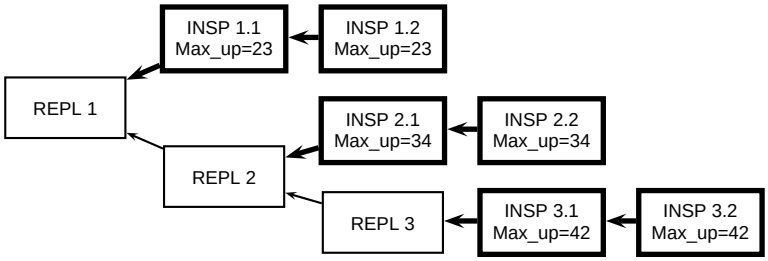


Figure 10.5: Inspections and their dependencies.

Opportunities

An opportunity is a time point at which maintenance can be done. In PM-opt, all opportunities must be specified, since the optimizer will not plan maintenance on a date which is not an opportunity. An opportunity consists of a date, a maximum work duration at the opportunity (its capacity), a base cost for usage (deducted as soon as at least one maintenance item is allocated to the opportunity), a downtime cost (deducted for each downtime minute of the opportunity) and a specification of the number of labor shifts to use. This forms the basis for computing the customer costs of maintenance.

10.3.3 Maintenance Package Planning

In traditional maintenance planning, maintenance packages each containing a set of maintenance activities for a set of components are predetermined and preplanned. The situation changes when the current condition, and therefore the different maintenance needs of the individual components of the turbine, is known. The problem is to decide which maintenance activities should be grouped together at which point in time, thus forming a dynamic maintenance package. For each package, the component to be serviced first limits the latest date at which the maintenance package can be performed. In PM-opt, each maintenance item is associated with a turbine stop time (for stopping and cooling down the turbine and the like), a duration of the actual activities contained in the maintenance item, and a restart time (including tests performed before start). The allocation of a maintenance item also gives rise to a cost proportional to its wasted lifetime. In addition, each maintenance item has a cost associated with the labor and material used. Costs due to increased labor level are modeled by an individual multiplier for each labor level. Each potential

maintenance stop is also associated with a base cost for using the stop and a variable cost that is associated with the actual downtime of the stop.

Maintenance Optimization Algorithm

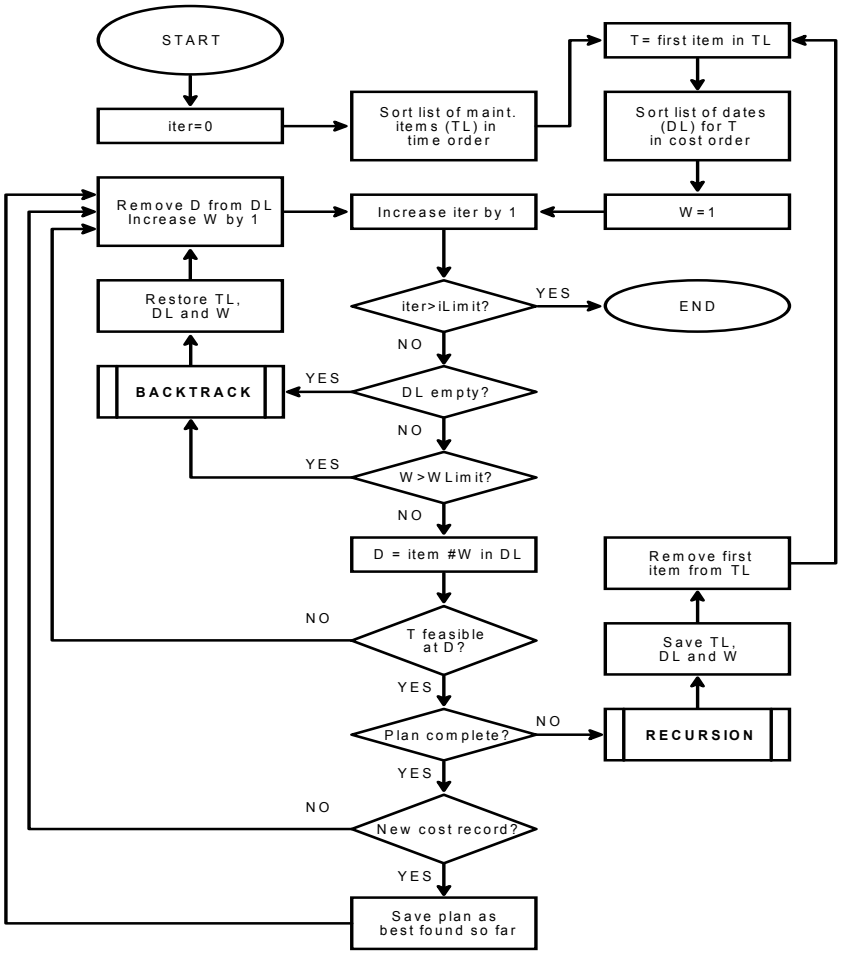
The optimization algorithm used is divided into two stages. The first stage (shown in Alg. 10.1) applies a depth-first branch-and-bound algorithm [11] augmented by a heuristic to find the best possible maintenance plan. On top of this, an iterative widening technique called Limited Discrepancy Search [6] is used to find better maintenance plans fast. In short, the first stage selects individual maintenance activities in turn, ordered by increasing deadline. The algorithm then tries to allocate the selected maintenance activity to each possible opportunity, and evaluates the results of doing this in combination with the previously-committed allocations. The possible opportunities are then sorted according to increasing cost, and the best one is committed for further evaluation. The search then proceeds by selecting the next activity in turn for allocation. If at any instant an inconsistency is detected, the search backtracks to the previous choice, and the next best opportunity is tried instead. At each allocation choice, k opportunities at the most are evaluated; wherever no opportunities exist, the search backtracks. In the experiments, it was found that iteratively increasing k from 0 to 2, resolving the problem for each k, provides good performance and quickly returns feasible improving solutions.

When all activities have been allocated to opportunities, the cost of the resulting plan can be established. If the generated plan is better than the best one found at that point, it is saved for future reference. The search then backtracks in order to find better plans. Also, a simple, lower bound of the cost of an incomplete plan is computed and used to backtrack as soon as the lower bound is higher than the cost of the best plan found at that point.

The second stage of the algorithm is similar to the first. One difference, however, is that a breadth-first technique similar to A* search [11] is used (instead of a depth-first strategy). In this stage, a set of partial plans ("nodes") are kept ordered according to a heuristic value, which is the sum of the partial plan cost and the lower bound for the unallocated maintenance activities. The node with the lowest heuristic value is selected first, and is expanded into several new nodes, which are inserted into the set. The expansion is done by splitting the set of possible opportunities for an activity into two sets. This yields two new nodes, each having one part of the opportunity set for the activity.

Since planning an activity at an early point in time incurs a cost in lifetime loss for the corresponding component, the optimizer favors late maintenance

Algorithm 10.1: Optimization algorithm (stage 1).



dates over early ones. However, since downtime costs are also taken into consideration, it may be even more favorable to place an activity at an earlier date. This is true if it already contains other maintenance activities (in which case the downtime increase will be smaller or even zero) or if the downtime has a significantly lower cost at the earlier opportunity. It was discovered that this simple heuristic works very well in practice and in many cases cannot be improved

upon. In fact, in the case where each opportunity has an equal downtime cost, the optimizer closely mimics the behavior of a human planner trying his/her best to use the components to their maximum.

The A* algorithm is guaranteed to find the optimal solution given enough time. However, since the search space is huge, finding the optimal solution is not plausible. In practice, the first stage of the algorithm provides a solution with sufficiently high quality for most purposes within a short response time, mainly due to the heuristics employed.

Problem Complexity

As a note on the problem difficulty from an optimization perspective, the planning problem can be seen as a generalized and more complicated form of bin-packing [1, 4], a well-known difficult problem in which the optimization time in practice is proportional to an exponential function of the problem size [8]. For this and other reasons, bin-packing problems are usually solved only to approximation using heuristics. This has turned out to be a suitable choice, since the heuristics used produce good solutions within an execution time of seconds.

In [14], maintenance planning is done for a more restrictive system where certain mathematical properties of the cost function must hold and where the potential gain of co-allocating maintenance is constant for all activities. In our model, cost is a function of component and downtime costs. This makes our model more expressive and, thus, not solvable using the polynomial solution approach in [14]. However, the heuristic used guarantees that the same solutions are explored for problems where the restrictions in [14] hold.

10.4 The Gas Turbine Maintenance Process

A strategy for gas turbine maintenance planning is proposed that consists of online lifetime prediction using a prognostics tool and online maintenance optimization using PM-opt. The workflow during CBM includes the following steps:

1. Identification of single and composed components and corresponding activities (maintenance items). If a maintenance activity can belong to more than a single component, it is either split into separate activities for the components in question or added as an activity for a generic placeholder component for “hard to specify” maintenance, which spans several components.

2. Lifetime predictions using a prognostics tool for suitable components.
3. Lifetime predictions using regular EOH/EOC estimates for components without suitable damage accumulation algorithms.
4. Construction of maintenance requirements. In this phase, a maintenance specification is entered into PM-opt. The specification consists of requirements that have to be fulfilled in all maintenance plans that will be constructed, and includes the maximum lifetime of the components in the gas turbine.
5. Contract period specification. PM-opt needs to know how long the gas turbine maintenance schedule should be.
6. Opportunity specification. In this phase, the loss of production costs and their corresponding calendar time intervals for the contract period are specified.
7. Initial maintenance planning, where PM-opt is used to optimize maintenance and get an initial maintenance plan.
8. Lifetime revisions using a prognostics tool. At inspections or when condition data has been read from the gas turbines, new lifetimes are computed and inserted into PM-opt.
9. Handling of unexpected events. These are inserted manually as unplanned activities into the maintenance schedule, together with suitable adjustments on remaining lifetime of the components affected.
10. Handling production changes. Production changes mean that the future opportunities, costs and priorities change, and such changes must be fed into PM-opt in order to produce relevant results.
11. Handling other changes in the maintenance plan. When the maintenance plan of the gas turbine is changed for other reasons not part of the CBM process, PM-opt still needs new maintenance schedules, so that maintenance is still planned according to the current state of practice.
12. Date adjustment. PM-opt needs to know which part of the maintenance plan has already been executed and, consequently, which activities are still pending (and thus needs to be scheduled).
13. Repetition of steps 8-12 until the maintenance contract is finished.

10.5 Evaluation

In this section, the approach presented in this paper is evaluated using a real world scenario in the oil and gas business. The gas turbine used for evaluation (a Siemens SGT-600) consists of 17 components with individual maintenance schedules. For some of the components, maintenance deadlines were determined from predictive lifetime analyses using a prognostics tool. Other components in the gas turbine were required to be maintained according to their original maintenance schedules.

10.5.1 Setup

The standard maintenance schedule, which is used for comparison, is based on the EOH/EOC calculations described in Eqn. (10.1) in Sect. 10.2.1. For the gas turbine in question, $C_x(t)$ was on average 0.72, and the other factors were always 1.0.

The critical components in the gas generator stage (combustion chamber, burners, compressor turbine guide vanes, and blades) were modeled and evaluated in a prognostics tool to determine suitable inspection intervals. However, at the time of writing, lifetime data was not available for the combustion chamber and burner components. Therefore, the original maintenance deadlines were used for these components. For the critical components, the relative increases obtained from using the prognostics tool, compared to the standard maintenance schedule, are shown in Tab. 10.1. The approach was validated as described in Sect. 10.5.3.

In Tab. 10.1, replacements marked *n/a* were not present in the EOH/EOC schedule, and are therefore not included in the prognostic schedule. Replacements marked *n/n* were not necessary in the prognostic schedule, since the estimated component lifetimes were significantly higher than the standard lifetime of the turbine.

Maintenance Scenarios

A standard maintenance contract of 15 years was assumed, for which maintenance should be optimized with regard to both maintenance costs and costs due to loss of production. The maintenance deadlines were used as the basis for computing suitable maintenance packages (and schedules) using PM-opt. The resulting schedules were analyzed with regard to 1) cost of production losses and 2) maintenance costs. PM-opt was set to run for 30 seconds at the most, producing the best maintenance schedule found within this time period.

Table 10.1: Increases in maintenance intervals obtained from the prognostics tool.

	Prognostics	
Component	Inspection	Replacement
Guide Vanes Stage 1	88 %	*n/a*
Guide Vanes Stage 2	151 %	*n/n*
Combustion Chamber	0 %	0 %
Burners	0 %	0 %
Blades Stage 1	101 %	*n/n*
Blades Stage 2	41 %	*n/n*
Blades Stage 3	245 %	*n/n*
Blades Stage 4	72 %	*n/n*

The evaluation was done on two scenarios. In the first scenario, high costs for lost production were assumed. The exception was for a three week period during the summer, where maintenance could be done without any negative effects on production. Such opportunities for maintenance with reduced or negligible negative effects on production are common in practice. In the second scenario, no such favorable opportunities were made available to the optimizer. In both scenarios, a low cost was associated with all maintenance stops. This cost corresponds to shutdown and startup costs. Maintenance was assumed to be done using one single shift of labor only.

10.5.2 Results

In the evaluation, four different maintenance strategies for two situations in two different scenarios were compared. The scenarios were set up to simulate maintenance planning for a new gas turbine and for a gas turbine with a non-empty maintenance history, in the cases where a seasonal stop was either absent or present. The four maintenance strategies were set up to simulate either the absence or the presence of advanced prognostics and/or maintenance optimization respectively.

New Gas Turbine

Table 10.2 shows results for a simulated brand new gas turbine. Since a new gas turbine should have an empty maintenance history, all component lifetimes are set to their predicted values. Where lifetimes are obtained from the standard maintenance schedule for the gas turbine, the necessary maintenance time points are already synchronized according to the maintenance packages determined for the original maintenance schedule. This makes the planning of maintenance packages easier, especially in the beginning of the contract.

In Tab. 10.2, the rows "EOH" and "Progn" give the results for schedules obtained by planning maintenance activities at the last possible date, given by the maintenance intervals obtained from standard EOH calculations and the prognostics tool respectively. This corresponds to the theoretically best possible case from a direct maintenance perspective (in other words, not considering the effect on the customer) and is obtained without using any minimization of production losses. On the other hand, the rows marked "EOH opt" and "Progn opt" provide results for schedules obtained by optimizing maintenance with regard to both maintenance and customer (loss of production) costs. Results are given for two scenarios; one where there is an already preplanned production stop of three weeks during the summer ("With seasonal stop") and one where production is assumed to continue throughout the year, without advantageous maintenance opportunities ("Without seasonal stop"). In this second case, maintenance can be freely placed. However, since maintenance stops always incur a significant cost, more focus must be placed on grouping maintenance activities together in suitable packages.

In Tab. 10.2, results are reported in the form of availability ("Avail"), maintenance costs ("Maint index") and productive days spent doing maintenance ("DT days"). Availability is computed as the number of productive days when maintenance is *not* done (not including seasonal stops, which are assumed to be unproductive) divided by the total number of productive days for the maintenance contract. Direct maintenance costs include material and work costs. Maintenance costs are expressed using an index. In it, 100 represents the cost of doing maintenance according to the maintenance intervals computed using the EOH/EOC calculations in Eqn. (10.1), the current state of practice. The highlighted line in Tab. 10.2 is the reference case corresponding to maintenance being done at the latest possible date.

As can be seen in Tab. 10.2, better lifetime estimates had a significant result on maintenance costs, availability and downtime. Adding the optimization of maintenance (with regard to both maintenance costs and production losses)

Table 10.2: Results of maintenance optimization for a new gas turbine.

	With seasonal stop			*Without seasonal stop*		
	Avail %	Maint index	DT days	Avail %	Maint index	DT days
EOH	***97.60***	***100***	***131***	***97.60***	***100***	***131***
EOH opt	99.99	109	0.42	98.15	120	101
Progn	98.20	61	98	98.20	61	98
Progn opt	100.0	62	0	98.81	75	65

yields even better results, and increases direct maintenance costs slightly. This is natural, since production losses in this care are very costly and optimization is done with regard to both loss of production costs and direct maintenance costs. Table 10.2 also shows that for a schedule with no advantageous opportunities, downtime can be reduced by more than 50 % using PM-opt and a prognostics tool.

Used Gas Turbine

Table 10.3 shows the same scenario as that used in the previous section, but for a simulated gas turbine that is assumed to already be in use. The scenario is simulated by randomizing the initial state of the gas turbine components. The already-used lifetimes of the gas turbine components were approximated by a random number drawn from a uniform distribution between 0 and the maintenance interval for the component.

As expected, Tab. 10.3 shows in general higher costs and lower availability than Tab. 10.2 due to a more spread out maintenance need. Using a prognostics tool and PM-opt in this scenario also yields significant results. Downtime can be reduced by 65 % for a schedule with no advantageous opportunities, compared to the current state of practice. In the case where seasonal opportunities are present, downtime can be reduced from 259 to 11.6 days.

10.5.3 Validation

The extended lifetimes shown in Tab. 10.1 are estimates that need to be validated in practice. A partial validation has been done in that a reference gas

Table 10.3: Results of maintenance optimization for a gas turbine with randomly chosen history.

	With seasonal stop			*Without seasonal stop*		
	Avail %	Maint index	DT days	Avail %	Maint index	DT days
EOH	95.26	121	259	95.26	121	259
EOH opt	99.56	133	24.0	97.49	149	137
Progn	96.03	79	217	96.03	79	217
Progn opt	99.79	82	11.6	98.35	85	90

turbine that had operated under the same conditions used for the lifetime predictions was dismantled and thoroughly inspected for the accumulation of typical damage mechanisms, such as creep and oxidation. The turbine was dismantled after 20 000 EOH, which is the standard maintenance interval for type SGT-600. The analysis showed that the accumulated damage was significantly less than predicted using the EOH/EOC calculations in Eqn. (10.1). However, final validation has to wait until one or more reference gas turbines have been dismantled after a longer operational period.

The PM-opt software is currently in use for gas turbine maintenance planning at Siemens Industrial Turbomachinery AB. Calculations show a possible increase in availability of 0.5 to 1 % in practice. Even when better lifetime predictions are not available and maintenance intervals are kept at the same length as before, significant reductions of preventive maintenance downtime are possible. Reductions of downtime by approximately 12 % due solely to the improved planning of preventive maintenance activities are expected in the general case.

10.6 Conclusions

In this paper, a new condition-based maintenance strategy was described wherein lifetime predictions and maintenance optimization are combined to safely decrease the maintenance costs, both direct and indirect, for gas turbines. The potential of the proposed approach was demonstrated using a real-world scenario, where maintenance planned using conventional methods was compared

with plans obtained from the process described in this paper. The approach was validated through the inspection of a reference gas turbine after a predetermined time interval. The results in this paper show that maintenance costs may be decreased substantially by using better lifetime predictions (based on customer requirements and the production plan) and by using maintenance optimization, minimizing maintenance downtime and production losses. In the simulations, direct maintenance costs were reduced by 25 to 38 %, while at the same time costs associated with downtime were reduced by typically 50 to 65 %. Finally, calculations regarding the effects of using the approach in this paper in practice show a possible increase in availability of 0.5 to 1 %, and expected downtime reductions of approximately 12 % due solely to improved planning. Both represent significant improvements.

Acknowledgment

This work was funded by the Swedish Institute of Computer Science and Siemens Industrial Turbomachinery AB.

Bibliography

[1] J. M. V. DE CARVALHO, *LP Models for Bin Packing and Cutting Stock Problems*, European Journal of Operations Research, 141 (2002), pp. 253–273.

[2] H. DEPOLD AND J. SIEGEL, *Using Diagnostics and Prognostics to Minimize the Cost of Ownership of Gas Turbines*, in *Proceedings of the ASME Turbo Expo*, 2006. Paper no. GT2006-91183.

[3] R. FRIEND, *A Probabilistic, Diagnostic and Prognostic System for Engine Health and Usage Management*, in IEEE Aerospace Conference Proceedings, vol. 6, March 2000, pp. 185–192.

[4] A. S. FUKUNAGA AND R. E. KORF, *Bin-Completion Algorithms for Multicontainer Packing and Covering Problems*, in Proceedings of the 19th International Joint Conference on Artificial Intelligence, L. P. Kaelbling and A. Saffiotti, eds., Professional Book Center, July 2005, pp. 117–124.

[5] J. A. HARRIS, *Engine Component Retirement for Cause*. AFWAL-TR-87-4069, Vol. 1, August 1987.

[6] W. D. HARVEY AND M. L. GINSBERG, *Limited Discrepancy Search*, in Proceedings of the 14th International Joint Conference on Artificial Intelligence, 1995, pp. 607–615.

[7] J. J. MCGROARTY AND J. W. SCHARSCHAN, *Integration of Condition Based Maintenance Technologies into U.S. Navy Gas Turbine Engines*, in Proceedings of the ASME Turbo Expo, 2002. Paper no. GT2002-30676.

[8] C. M. PAPADIMITRIOU, *Computational Complexity*, Addison-Wesley, Reading, Massachusetts, 1994.

[9] R. POOL, *"If It Ain't Broke, Fix It"*. Technology Review, September 2001.

[10] R. RAJAMANI, J. WANG, AND K. Y. JOENG, *Condition Based Maintenance for Aircraft Engines*, in *Proceedings of the ASME Turbo Expo*, 2004. Paper no. GT2004-54127.

[11] S. RUSSELL AND P. NORVIG, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Second ed., 2003.

[12] P. SLOTTNER AND M. WÄRJA, *Knowledge Based Prognostics Models for Gas Turbine Core Components*, in *Proceedings of the ASME Turbo Expo*, 2008. Paper no. GT2008-51276.

[13] S. VITTAL, P. HAJELA, AND A. JOSHI, *Review of Approaches to Gas Turbine Life Management*, in *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, vol. 2, 2004, pp. 876–886.

[14] R. E. WILDEMAN, R. DEKKER, AND A. C. J. M. SMIT, *A Dynamic Policy for Grouping Maintenance Activities*, European Journal of Operations Research, 99 (1997), pp. 530–551.

[15] M. WÄRJA, P. SLOTTNER, AND M. BOHLIN, *Customer Adapted Maintenance Plan (CAMP), a Process for Optimization of Gas Turbine Maintenance*, in *Proceedings of the ASME Turbo Expo*, 2008. Paper no. GT2008-50240.

Chapter 11

Paper F: Scheduling Gas Turbine Maintenance Based on Condition Data

Markus Bohlin, Kivanc Doganay, Per Kreuger, Mathias Wärja and Rebecca Steinert.

In Proceedings of the 21st Innovative Applications of Artificial Intelligence Conference.

July 14–16, 2009, Pasadena, California, USA.

Abstract

We describe the implementation and deployment of a software decision support tool for the maintenance planning of gas turbines. The tool is used to plan the maintenance for turbines manufactured and maintained by Siemens Industrial Turbomachinery AB (SIT AB) with the goal to reduce the direct maintenance costs and the often very costly production losses during maintenance downtime. The optimization problem is formally defined, and we argue that feasibility in it is NP-complete. We outline a heuristic algorithm that can quickly solve the problem for practical purposes, and validate the approach on a real-world scenario based on an oil production facility. We also compare the performance of our algorithm with results from using mixed integer linear programming, and discuss the deployment of the application. The experimental results indicate that downtime reductions up to 65% can be achieved, compared to traditional preventive maintenance. In addition, using our tool is expected to improve availability with up to 1% and reduce the number of planned maintenance days with 12%. Compared to a mixed integer programming approach, our algorithm not optimal, but is orders of magnitude faster and produces results which are useful in practice. Our test results and SIT AB's estimates based on operational use both indicate that significant savings can be achieved by using our software tool, compared to maintenance plans with fixed intervals.

11.1 Introduction

Preventive maintenance can reduce breakdowns and costs associated with them, but is also costly when done frequently. That is why considerable effort (e.g. [3, 10]) has previously been spent on optimizing maintenance so that the expected total cost due to failures and preventive maintenance is minimized. Most preventive maintenance approaches use fixed schedules, which are optimized for minimum cost in advance. However, there are many situations in which maintenance re-planning is in practice necessary to being able to continue operation and to lower costs. For example, unexpected breakdowns force the production unit to stop for emergency repair, and it would be unwise not to consider performing other maintenance tasks at the same time. Other examples include production stops for other reasons than maintenance, which provides valuable opportunities for maintenance. The introduction of condition monitoring has also lead to the replacement of preventive maintenance with condition-based corrective maintenance, which is by nature less predictable than a fixed preventive maintenance plan.

In this paper, we present the ideas behind a tool, PMOPT (Preventive Maintenance Optimizer), for gas turbine maintenance planning. PMOPT was developed for Siemens Industrial Turbomachinery AB (SIT AB), one of the leading manufacturers of gas turbines of small and medium size. Gas turbines are used for power-generation in various production facilities that often have high downtime costs. A typical gas turbine application is offshore oil platforms, where time spent without power can cause an extremely high loss of revenue. In such applications, small improvements in terms of overall availability, which is one expected outcome of implementing CBM, have a substantial positive effect on the total income for the customer.

Condition-based gas turbine maintenance, where component lifetime is dependent on factors such as load profile, quality of fuel, ambient temperature, and particle levels, is becoming more and more common. Although lifetime predictions can sometimes be performed with high precision, maintenance time points will still vary depending on the conditions on site, and the actual time points will therefore also diverge from their original estimates.

The approach presented in this paper aims at providing a tool which can quickly optimize maintenance when unplanned events make the current maintenance schedule unsuitable. We use a rolled-out representation of a deterministic future maintenance schedule, which makes it possible to take into account positive effects of co-allocation, maintenance opportunities, overall availability, horizon effects and costs due to both maintenance and loss-of-production.

Proper risk analysis and deterioration model identification can in many practical cases be difficult to perform from scratch. As a consequence, maintenance intervals are often based on analytical models and "best practice". Instead of using failure rate distributions to make tradeoffs between costs for breakdown and preventive maintenance, we therefore assume a safe deadline for maintenance activities, which simplifies the problem and makes it easy to adapt already existing maintenance plans for use in PMOPT.

The contributions of this paper include that we 1) precisely define the maintenance scheduling problem discussed, 2) argue that the planning problem is NP-complete, 3) outline an algorithm that can quickly solve the problem for practical purposes, 4) show results for a real-world scenario, 5) compare the results of our algorithm to the results from using mixed integer linear programming, and 6) discuss the implementation and deployment of PMOPT.

11.1.1 Related Work

Maintenance optimization is certainly not a new topic. An excellent overview of the many applications considered can be found in [3]. In [12] dynamic programming is used to optimize maintenance planning with respect to acceptable equipment reliability, demand of generating units and maintenance cost. However, overall availability as a fraction of the total time is not considered, and the crew and resource model used does not consider downtime due to day and week rest.

Other approaches to maintenance optimization are based on Monte Carlo simulations combined with genetic algorithms [8]. In a related approach described in [10], pre-planned maintenance opportunities are taken into account similarly to our own method. However, their approach is non-deterministic in contrast to our optimization method.

In [11], maintenance planning is done for a more restrictive system where certain properties of the cost function must hold, and where potential gain of co-allocating maintenance is constant for all activities. In our model, cost is a function of component costs and indirect costs, resulting from unavailability of the gas turbine due to maintenance. This makes our model more expressive, and thus unsolvable using the polynomial solution approach in [11].

11.2 Background

The common practice of gas turbine maintenance planning today is to base the schedules on Equivalent Operating Hours (EOH) and cycles (i.e., the number

of turbine restarts). The number of operating hours is modified with factors for load, fuel quality, presence of water injection, and (to a limited extent) significant exhaust temperature differences. However, the model is not detailed enough in how these variables are handled, and factors such as ambient air temperature and pressure, rotational speeds, and more detailed outlet temperatures are not included. Instead, the EOH calculations have substantial built-in safety margins to accommodate for variables not explicitly modeled.

In order to improve overall maintenance efficiency, new calculations for estimating the remaining lifetime of gas turbine components based on operation profile, environmental conditions, and condition data obtained through inspections and sensors on the gas turbine has been developed by SIT AB. A lifetime prediction tool, producing deterministic lifetime estimates, has also been developed. The lifetime estimates produced by the tool include relevant safety margins. Therefore, changes in lifetime should not affect risk levels negatively as long as the gas turbine is serviced within its predicted lifetime. In fact, risk levels can in many cases be dramatically reduced, since the lifetime prediction tool also detects and decreases maintenance intervals for gas turbines operating under conditions with increased component wear (for example high load, high humidity or low fuel quality).

11.2.1 Improved Analytical Lifetime Predictions

Gas turbine component lifetime is to a great extent determined by operation temperatures. However, it is also determined by the extreme rotational load and pressures that some parts are exposed to. The gas turbine cycle is also highly sensitive to ambient conditions (mainly inlet air pressure and temperature). The following procedure is employed to calculate the component lifetime for a specific situation.

1. First, the overall energy balance of the gas turbine is calculated using heat balance evaluations based on measurements of pressures, temperatures and rotational speeds at various locations in the gas turbine.

2. Based upon the energy balance input, we then calculate the expected mass flow, temperature and pressure at locations where sensors cannot easily be placed or hamper performance (such as within the hot gas pass, inside the combustion chamber, and inside the rotors). The calculations are performed using standard methods from combustion kinetics, aerodynamics, flow distribution and cooling codes.

3. Finally, we compute the mechanical response to the thermal, aerodynamical and mechanical loads for component sets that interact mechanically with each other.

The results of step 2 and 3 are then used to compute an expected lifetime. Some of the involved calculations are carried out using the finite element method (see for example [13]). However, the applied fluid and solid models are specifically adapted to gas turbine conditions and materials.

Calculation time for the process outlined above can range from weeks to months per iteration. Therefore, a pre-computed approximation is used for real-life prediction. The approximation is refined by manual correction using experience from service and risk assessments to be accurate enough and to provide sufficient safety margins.

11.3 Problem Description

In this section, we first give an informal description of the scheduling problem that PMOPT is aimed at solving. We then define the duration model adopted in this paper, which includes calculations of total work and stop time for an maintenance stop. This is followed by a more rigorous definition of the scheduling problem we want to solve. The section ends with an argument for why feasibility in the maintenance scheduling problem is NP-complete, and why new solution methods are needed to solve it.

We can informally describe the Maintenance Scheduling with Opportunities Problem (MSOP) as the problem of allocating maintenance *items* to dates for k independent components in a single unit and for a time period of h weeks, so that constraints on timeliness, work time capacity and total availability are satisfied. The allocation should minimize direct and indirect maintenance costs, including spare parts, labor, and value of production lost due to maintenance.

Each component has a cyclical schedule of arbitrary length, consisting of *inspections* and *replacements*. The date of a replacement depends only on the previous replacement, while inspections depend on the previous item regardless of type. We assume that the obtained lifetime estimates used as input to the optimizer are safe in the sense that if maintenance deadlines are met, risk levels are negligible. Also, we assume that the given component schedules are followed and that deviations are taken into account by updating the schedule data. The problem is therefore deterministic in nature.

11.3.1 Duration Models

To estimate work time at a maintenance stop, each maintenance item has a *duration specification* $\Delta_i = \langle \Delta_{1i}, \Delta_{2i}, \ldots, \Delta_{Bi} \rangle$ divided into non-negative *work phases* Δ_{bi}, where at least one phase has to be non-zero. The set of work phases are denoted by $\mathcal{B}$. All items with activities within a single phase at a single stop are assumed to be fully independent, and can therefore be executed in parallel. In contrast, the phases themselves have to be done in an orderly fashion, and therefore have to be executed serially. The total work time u_j of a stop can thus be computed as the sum of the maximum work time in each block.

As an example, consider the two duration specifications $\langle 3, 1, 0, 5 \rangle$ and $\langle 4, 0, 2, 3 \rangle$ allocated to the same stop. The working time for the different phases then becomes $\langle 4, 1, 2, 5 \rangle$, and the total work time at the stop is 12.

Given the total work time at a stop, we can now compute the *downtime*. We assume that a working day consists of A hours, and that all calendar weeks (consisting of 6 working days) are alike. The downtime of non-empty stops is computed by adding night-rest time for each day when all work was not finished, and week-rest time for each week when all work was not finished, using the following function.

$$\mathcal{D}(W) = W + (24 - A)\left\lceil \frac{W}{A} - 1 \right\rceil + 24\left\lceil \frac{W}{6A} - 1 \right\rceil \tag{11.1}$$

For empty stops, $\mathcal{D}(W) = 0$. In the rest of this paper, we assume that $A = 10$.

11.3.2 Optimization Model

We assume that n maintenance items denoted by $i \in \mathcal{I}$ have been rolled out to cover weeks 1 to h (the *horizon* of the problem). The decision variable $\mathbf{t}[i]$ represent the date of item i. The schedule end is modeled by the artificial item $\top$ at date $h + 1$, and the schedule start is modeled by another artificial item $\bot$ at date 0. The possible allocation dates within the schedule are modeled by a finite set $\mathcal{O}$ of *opportunities* j with dates δ_j and work time capacity v_j.

Timeliness constraints are expressed as follows. Each item i has a *release time* o_i and a *deadline* d_i relative to i's *predecessor* p_i. Each item also has an optional *earliest* and *latest date* of execution ($t_i^{\min}$ and $t_i^{\max}$). We assume that each replacement for a component starts a new sequence of inspections, which makes items from previous sequences redundant. We call rolled-out items that do not have to be executed *obsolete* items.

Each item i has a *terminator* s_i that makes i obsolete if i is done later or at the same date as s_i. For simplicity, we force obsolete items to be allocated to the same date as their terminator. Formally, we define the predicate $\mathrm{obs}(i)$, with the meaning that activity i is made obsolete by its terminator s_i, as follows.

$$\mathrm{obs}(i) \equiv \mathbf{t}[i] = \mathbf{t}[s_i] \tag{11.2}$$

Replacements always have $\top$ as their terminator, which implies that they are only made obsolete by being moved over the problem horizon h. The top of Figure 11.1 illustrates relative timeliness constraints (release times and deadlines) between pairs of tasks as well as predecessor and terminator relationships in a fictional schedule.

The first items in the schedule for each component is called the set of *head* items, and is denoted $\mathcal{E}$. All head items are assumed to have $\perp$ as their predecessor. To ensure that the gaps after sequences of items are not too large, we use special items representing the end of such sequences. We call such items *tail* items. The set of tail items $\mathcal{L}$ consists of 1) the last replacement for each component, and 2) the last item in each inspection sequence. By forcing all tail items to be obsolete, the normal deadline constraints ensure that end gaps are smaller than required for all feasible solutions. The concepts are illustrated in the bottom of Figure 11.1.

Each item also has an *item cost* c_i consisting of work and material cost. The value of production per hour at an opportunity j is denoted l_j. In addition, we use a fixed *base* cost (b_j) for opening up opportunity j. The base cost is associated with setup costs for shutting down and restarting the gas turbine, travel expenses, and other costs that cannot be modeled using material, work or downtime costs.

Minimum availability is specified by the user via the parameter α (where $0 \leq \alpha \leq 1$). The total availability is defined as the productive time not spent on preventive maintenance divided by the total available productive time. The constraints in the problem can now be stated.

- Each item i should be allocated to a date $\mathbf{t}[i]$ that is less than or equal to its deadline.

$$\forall i \in \mathcal{I} : \mathbf{t}[i] \leq \mathbf{t}[p_i] + d_i \tag{11.3}$$

- Each item has to respect its absolute allocation interval.

$$\forall i \in \mathcal{I} : t_i^{\min} \leq \mathbf{t}[i] \leq t_i^{\max} \tag{11.4}$$

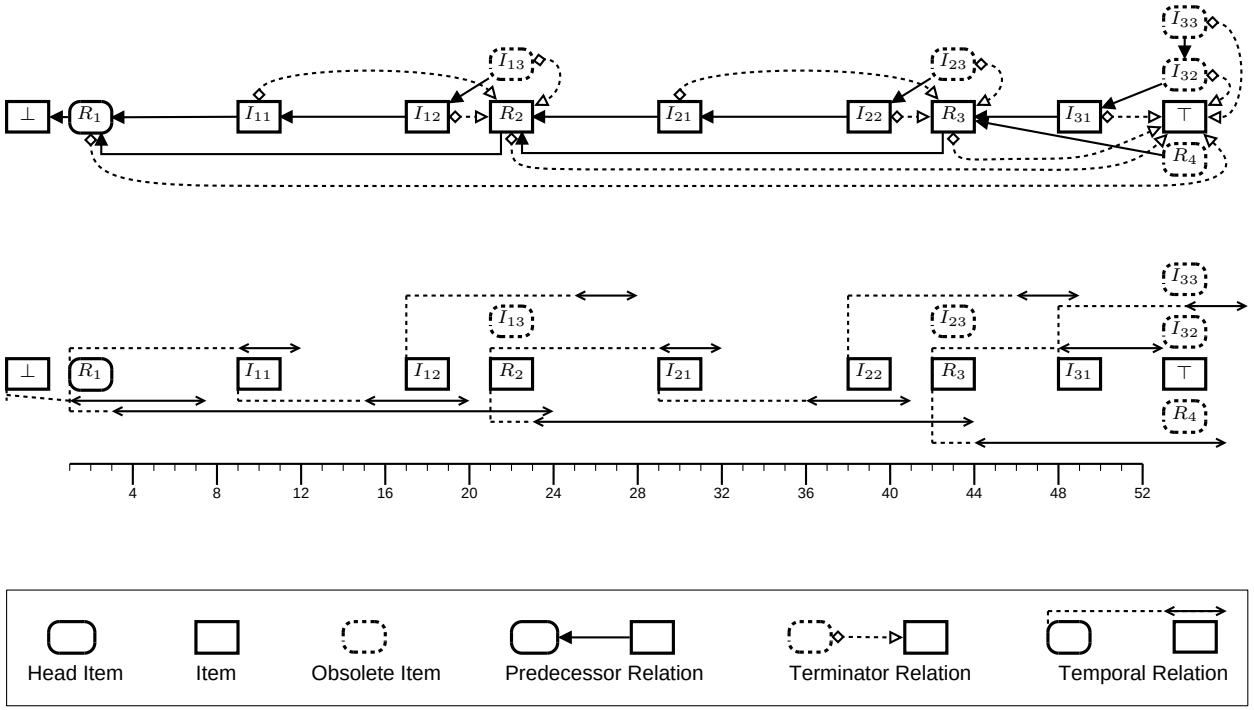


Figure 11.1: Dependencies (top) and relative timeliness constraints (bottom) between different maintenance activities of a component.

- Each tail item has to be obsolete.

$$\forall i \in \mathcal{L} : \mathrm{obs}(i) \tag{11.5}$$

- Each non-tail item i should be either obsolete or allocated to a date larger than its offset.

$$\forall i \in \mathcal{I} \setminus \mathcal{L} : \mathrm{obs}(i) \vee \mathbf{t}[i] \geq \mathbf{t}[p_i] + o_i \tag{11.6}$$

- For each opportunity j, the total work time u_j allocated to j must be lower than the capacity of j.

$$\begin{aligned} &\forall j \in \mathcal{O} : u_j = \sum_{b \in \mathcal{B}} \max_{\substack{i \in \mathcal{I} \\ \mathbf{t}[i] = \delta_j \wedge \neg \mathrm{obs}(i)}} \Delta_{bi} \\ &\forall j \in \mathcal{O} : u_j \leq v_j \end{aligned} \tag{11.7}$$

- The availability of the plan should be greater than the minimum availability α.

$$\frac{1}{7 \cdot 24} \sum_{\substack{j \in \mathcal{O} \\ \exists i \in \mathcal{I} : \mathbf{t}[i] = \delta_j}} \mathcal{D}(u_j) \leq h(1 - a) \tag{11.8}$$

The objective of the optimization problem is to minimize the cost function f, defined as follows.

$$f(\mathbf{t}) = \sum_{\substack{i \in \mathcal{I} \\ \neg \mathrm{obs}(i)}} c_i + \sum_{j \in \mathcal{O}} l_j \mathcal{D}(u_j) + \sum_{\substack{j \in \mathcal{O} \\ \exists i \in \mathcal{I} : \mathbf{t}[i] = \delta_j}} b_j \tag{11.9}$$

The first term is the maintenance cost of all items within the horizon, the second term is the indirect costs for the opportunities, and the third term is the base costs.

11.3.3 Complexity

Feasibility in MSOP (or FMSOP for short), that is, the question whether any feasible solution to MSOP exists, is NP-complete. We argue in this section that 1) FMSOP is in NP by outlining a polynomial-time verification algorithm

([2]), and 2) that there is a polynomial-time reduction from the bin packing problem (BPP; see for example [4]), to FMSOP.

The objective of BPP is to pack items $i \in \{1, \ldots, n\}$ of given sizes a_i into as few bins (with fixed capacity V) as possible. The used capacity of a bin is computed as the sum of the weights of the items in the bin. The decision variant of BPP answers the question whether a packing for any given number of bins B exists.

1. Given a candidate solution C to FMSOP (i.e. an assignment of dates to the maintenance items in C), we can verify the constraints on structure and timeliness by simply testing Equations (11.3), (11.4), (11.5) and (11.6) for the given dates of the item and its predecessor and terminator. This can be done in linear time to the number of items. The capacity constraints in Eqn. (11.7) can easily be verified by investigating the items allocated to that opportunity in time $O(nm)$. The availability constraint in Eqn. (11.8) can be verified in a similar way as for the capacity constraints. This together with computations of the downtime function in Eqn. (11.1) can be done in time $O(nm)$. The procedure outlined above is clearly polynomial, and therefore FMSOP is in NP.

2. We can translate a given BPP into a FMSOP by having B opportunities, each opportunity j (where $1 \leq j \leq B$) having date $\delta_j = j$ and capacity $v_j = V$. Let the horizon $h = B + 1$. Each BPP item i is translated into a FMSOP replacement item i with $\perp$ as predecessor, 0 as release time, B as deadline, $t_i^{\min} = 0$ and $t_i^{\max} = h$. The duration $\Delta_{bi} = a_i$ if $b = i$ and 0 otherwise, i.e., the duration (weight) of an item is always put in a unique phase in Δ_i. All items i have an artificial item $n+i$ as terminator, which in turn have release time 1, deadline $h + 1$, $t_i^{\min} = 0$, $t_i^{\max} = h + 1$, $\top$ as terminator and arbitrary duration. By definition, the tail items, being replacements, have to occur at $\top$, which is outside the schedule. Let the minimum availability requirement $\alpha = 0.0$.

   The transformed problem corresponds directly to BPP, since 1) each BPP item is represented by a FMSOP replacement, 2) each BPP bin is represented by a FMSOP opportunity with unique date and equal capacity, and 3) the total duration of an opportunity is computed as the sum of the item durations at that opportunity, since all durations are in unique working phases, which corresponds directly to the sum of the weights of items in a bin in BPP. All other constructs of FMSOP are disabled and therefore do not constrain the solution, and therefore, BPP is a special case of FMSOP.

If we could find a solution to the transformed FMSOP using a polynomial time algorithm, we could then use that algorithm to solve BPP (which is NP-complete, see [5]) in polynomial time. This, together with the previous conclusion that FMSOP is in NP, shows that FMSOP is NP-complete.

Efficient polynomial-time approximations exist for the bin-packing problem; see for example [4]. However, MSOP differs in objective from BPP, and has complicating side constraints that are missing in BPP. For example, in MSOP, each opportunity (date) can have a different capacity, base cost and downtime cost. In BPP, a bin is defined only by its capacity, which is also uniform. Another difference is that items in MSOP can partially overlap within an opportunity due to the work time model used. This makes bin-packing approaches inapplicable to MSOP. It is currently an open issue whether polynomial-time approximation schemes exist for MSOP.

11.4 A Tool for Maintenance Scheduling

The optimization software consists of two separate programs that communicate using files; PMOPT-GUI and MAINTOPT. The architecture is shown in Figure 11.2. MAINTOPT is written in C++, and PMOPT-GUI is written in the C++/CLI programming language using the .NET platform. PMOPT does not require any special installation procedure; it simply runs as a stand-alone application on any computer where the .NET framework is installed.

The schedule and related information are considered to be a *project*, and is stored in a *project file*. A typical user would load a previously created project file directly after starting PMOPT. PMOPT-GUI makes it possible to edit the project file, and immediately displays the effects of edits, such as costs and availability. Edits include changing lifetime estimates, adding/deleting components and activities and moving/copying activities within and between components.

Whenever the user requests an optimization of the current maintenance plan, PMOPT-GUI produces a rolled-out representation of the specification, which is passed on to the optimizer. Time is translated into integer values, so that MAINTOPT does not need to be aware of the time scale. As soon as MAINTOPT finishes, the solution file is read back into PMOPT-GUI and shown to the user.

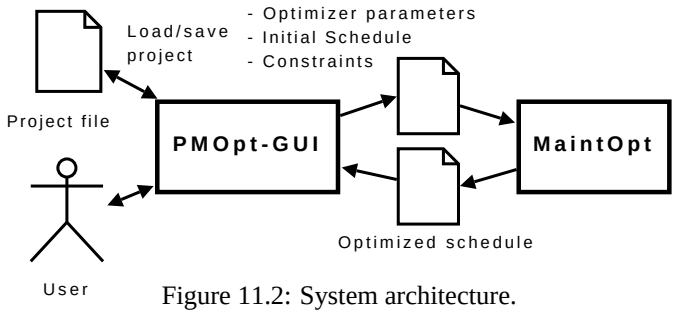


Figure 11.2: System architecture.

11.4.1 MAINTOPT *and the Optimization Algorithm*

The optimization algorithm should be able to produce maintenance schedules within a limited time in order to be used interactively. The optimization algorithm in MAINTOPT is based on Limited Discrepancy Search (LDS) [6]. Maintenance items are assigned in order of increasing deadline, and the value-selection heuristic picks opportunities in increasing cost order, with a bias for late opportunities. Only consistent assignments are considered; variable domains are pruned using interval propagation [7]. In our experiments, we have found that iteratively increasing the LDS width k from 0 to 2, resolving the problem for each k, gives overall good performance. The default optimization time is set to 30 seconds, which is more than enough for normal instances.

11.5 Development and Deployment

Manual planning is the norm in the gas turbine field, and before PMOPT and the lifetime prediction tool, SIT AB did not have any manual or automatic procedures for improving maintenance schedules. A standard schedule was used, which is equal to 20 000 operating hours and assumes a constant level of degradation at a standard pace for all components of the gas turbine. When the lifetime prediction techniques outlined in this paper had been developed, the need for maintenance planning in order to take advantage of possible lifetime extensions soon became apparent.

The Swedish Institute of Computer Science (SICS) was first approached by SIT AB regarding maintenance scheduling optimization during the summer of 2006 at an international conference related to condition monitoring. This first contact resulted in a sequence of meetings during the autumn with the purpose of evaluating the feasibility of the project idea. At this time point, the core

maintenance optimization engine (MAINTOPT) had already been developed for use in a different project in the railway domain. However, MAINTOPT was in its infancy, and it became obvious during our collaboration with SIT AB that we had to extend it with side constraints and objective function terms previously not considered. One example is the availability constraints and the focus on downtime as a critical parameter, which was not present in MAINTOPT at that time. However, being able to demonstrate the early version of the planning software together with demonstrator applications from previous projects helped a lot during these initial meetings.

Before starting the PMOPT development process, a commercial product for maintenance optimization had been evaluated at SIT AB. One of the main problems with the product was that it was not able to model important properties of the gas turbine planning problem, such as seasonal variations and usage profiles for different parameters like load, particle levels, and environmental factors. More importantly, generic tools often use costs based on statistics. In reality, it is not uncommon that one prefers not to use the statistically optimal point of lowest cost due to the need for safety margins. The consequences of some types of failures are also too severe to be estimated using statistical approaches. In addition, for a complex machine such as a gas turbine, it can be impractical to identify all possible failures, the corresponding statistical distributions, and all consequences and associated costs for each failure. Instead of having too many estimates, it was decided that a safe deadline for each maintenance activity was a better alternative.

SIT AB were heavily involved in the specification and development throughout the project, and this was a main factor behind the successful outcome. Without close collaboration with SIT AB, many details regarding the application area would have been missed due to lack of knowledge in that area. It would also have been difficult to motivate necessary design changes without support from our main contacts.

Throughout the project, five people from SICS were directly involved. We had two main contact persons at SIT AB, and several site managers were directly involved.

11.5.1 First Versions

In November 2006, SICS received a spreadsheet containing an early draft specification of the problem to be solved. The spreadsheet showed some ideas regarding calculations on downtime and maintenance activity packaging, and it was decided that a prototype should be developed from the draft specifica-

tion. The prototype was nothing more than a simple graphical front-end to MAINTOPT without any interaction. Nonetheless, it served the purpose of showing the feasibility of the project proposal well. After this pre-study and basic demonstration, we began discussions regarding the project economy and deliverables in early 2007. Soon after that the contracts were signed and development started. We finished the first release (version 0.9) in mid-April 2007. Due to time pressure, the first version ended up being more of a prototype than mature software. With many test releases in between, version 1.0 was finally shipped in June 2007.

From experience with the first releases, we soon realized that changes in the optimization engine were rather straightforward to implement. However, maintenance and extensions that primarily affected the management of the problem model proved to be much more time consuming. One of the biggest problems was to keep the model consistent and to handle the entire set of user actions and model parameters. For example, changes in the maintenance schedule made after running the gas turbine for some time needed to be synchronized with the already laid-out maintenance schedule up to the current time point. We soon realized that we had severely underestimated the work involved in managing the maintenance schedule. Other areas that needed more attention than expected were models of work time, application security, licensing, management of gas turbine maintenance projects, and user accounts and rights management.

11.5.2 Second Version

We made several changes to the basic design of PMOPT in the second phase of the project to simplify the maintenance of the application and facilitate future extensions. Rewriting the core of the application from scratch was perhaps the largest one, but significant changes were also made in the search algorithm. In the beginning, MAINTOPT was a pure branch-and-bound algorithm based on A* search [9]. However, after extensive experimentation with sample maintenance projects it became apparent that A* search, being based on breadth-first search, was spending too much time exploring high-level decisions in the search tree, and failed in finding feasible solutions quickly. Since responsiveness of the application was one of the main criteria of PMOPT, we resorted to experimentation with heuristics, and after a while, the LDS procedure was added as a first stage of the algorithm. Lately, the second stage A* search has been completely removed from MAINTOPT, since it does not really help in solving typical problem instances. In our experience, system responsiveness and producing a reasonably good solution fast was more important than pro-

ducing the absolute optimum. Tuning heuristics turned out to be an important task, as the standard A* and LDS algorithms were of limited value without guidance using the specific problem characteristics.

With major changes to the GUI and improved heuristics, a second major version (version 2.0) was released in March 2008. This version was delivered two months in advance of its deadline due to the much improved core design, which helped speed up the implementation of new features. Since then, more improvements have been made, with a new release in August the same year. The latest release (version 2.4) was shipped in November 2008.

11.5.3 Deployment at SIT AB

During the development of PMOPT, it became increasingly clear that a planning tool of this type is not easily deployed. First of all, key personnel need to be educated in the theories behind condition-based maintenance planning, and in how an automated tool can help in adjusting a schedule to customer-specific conditions. In addition, it was necessary to gain adequate insight into maintenance planning practices in order to increase the usability of the PMOPT tool. During the development of the first version, suggestions and ideas for the usability enhancement of the software were continuously discussed. Before deployment could begin, suitable business models also had to be developed, evaluations of current technology needed to be completed, personnel had to be trained in using the tool, and data acquisition routines, working processes and suitable information flows needed to be established. Therefore, PMOPT was not deployed in operational use until early 2008 after the release of version 2.0.

Currently, PMOPT is used by two people, mainly for planning of maintenance after deviations have occurred. PMOPT has been running operatively for verification/validation of the global CBM strategy for five months. It is used within two maintenance contracts; in the first, PMOPT is fully operational, while it is used for validation and testing purposes in the second one. Testing is done mainly for gaining feedback from practical experience, monitoring of environmental variables (e.g. temperatures), and time increments. In a couple of years, four or five people working within maintenance planning are expected to use the tools for 10–15 different operational contracts.

11.5.4 Application Maintenance and support

Maintenance of PMOPT were performed by SICS on demand when bug reports were filed, which happened mostly from our main contact people at SIT AB

after new releases had been shipped. Naturally, most bugs were reported just after the delivery of version 1.0.

Overall, larger improvements were mostly related to the GUI and the usability of the system. Current users have direct contact with us and are able to ask questions as well as request changes. During the development, our understanding of the domain improved and matured, and several improvements of the problem models were gradually implemented. Some changes were explicitly requested by SIT AB, while others were necessary to make the code base easy to maintain. As an example, the specification of the optimization model was changed several times, and the work time model, by request from SIT AB, was updated to more accurately capture the real duration, downtime and cost of a maintenance opportunity. Another change was proposed by the development team regarding the model of dependencies between maintenance items and the handling of obsolete items. The first model proposed was too simplistic in that there was no difference between inspections and replacements.

11.6 Estimated and Measured Benefits

In this section, PMOPT is evaluated on a real world scenario in the oil and gas business. The turbine under consideration has 17 components with individual schedules. A standard maintenance schedule for the site was used as a comparison. The critical components in the gas generator stage for which lifetime data was available (compressor turbine guide vanes and blades) were modeled and evaluated in a prognostics tool to determine suitable inspection intervals. The average increase in inspection time was 116 %, and replacements for the critical components were not necessary, since their predicted lifetime were much longer than the standard maintenance contract length of 15 years. The scenario is described in more detail in [1].

A partial validation of the obtained lifetimes has been done in that a reference gas turbine having operated under the same conditions was dismantled after a standard maintenance interval of 20 000 operating hours and thoroughly inspected. The analysis showed that the accumulated damage was significantly less than predicted using the standard EOH calculations. However, final validation has to wait until one or more reference gas turbines have been dismantled after a longer operational period.

The evaluation was done on two scenarios. The first scenario had a three week seasonal stop during the summer, where maintenance could be done without any negative effects on production. Such opportunities for maintenance are common in practice. In the second scenario, no such favorable opportunities

	With seasonal stop			*Without seasonal stop*		
	Avail %	Maint index	DT days	Avail %	Maint index	DT days
EOH	***97.60***	***100***	***131***	***97.60***	***100***	***131***
EOH opt	99.99	109	0.42	98.15	120	101
Progn	98.20	61	98	98.20	61	98
Progn opt	100.0	62	0	98.81	75	65

Table 11.1: Results of maintenance optimization for a new gas turbine.

existed. In both scenarios, a low base cost was associated with all maintenance stops, and high costs were associated with loss of production. The schedules resulting from running PMOPT were analyzed with regard to 1) cost of production losses and 2) maintenance costs. PMOPT was set to run for at most 30 seconds.

11.6.1 Results

Table 11.1 shows results for a simulated brand new gas turbine. The rows "EOH" and "Progn" correspond to planning maintenance at the last possible date, as specified using standard EOH calculations and the prognostics tool respectively. This approach minimizes direct maintenance costs while ignoring other costs. The rows marked "EOH opt" and "Progn opt" correspond to optimizing maintenance using PMOPT.

Results are reported in the form of availability ("Avail"), maintenance costs ("Maint index") and productive days spent doing maintenance ("DT days"). Maintenance costs are expressed using an index. In it, 100 represents the cost of doing maintenance according to the maintenance intervals computed using the standard schedule. In Tab. 11.1, this corresponds to the row typeset in boldface.

As can be seen in Tab. 11.1, better lifetime estimates had a significant result on maintenance costs, availability and downtime. Adding the optimization of maintenance using PMOPT yields even better results, and increases direct maintenance costs slightly. This is natural, since production losses in this case are very costly and optimization is done with regard to both loss of production costs and direct maintenance costs. Table 11.1 also shows that for a schedule with no advantageous opportunities, downtime can be reduced by more than 50 % using PMOPT and a prognostics tool.

	With seasonal stop			*Without seasonal stop*		
	Avail %	Maint index	DT days	Avail %	Maint index	DT days
EOH	95.26	121	259	95.26	121	259
EOH opt	99.56	133	24.0	97.49	149	137
Progn	96.03	79	217	96.03	79	217
Progn opt	99.79	82	11.6	98.35	85	90

Table 11.2: Results of maintenance optimization for a gas turbine with randomly chosen history.

	With seasonal stop			*Without seasonal stop*		
New turbine	Diff. %	Gap %	Time	Diff. %	Gap %	Time
EOH opt	-6.1	0	40m	–	∞	8h
Progn opt	-1.1	0	27m	$+93$	75.6	8h
Used turbine						
EOH opt	-23.6	1.79	8h	–	∞	8h
Progn opt	-0.6	0.95	8h	–	∞	8h

Table 11.3: Comparison of results between CPLEX 9.0 and PMOPT.

Used Gas Turbine

Table 11.2 shows the same scenario but for a simulated gas turbine already in use. The scenario is simulated by setting the already-used lifetimes of the gas turbine components to a random number drawn from a uniform distribution between zero and the maintenance interval for the component. As expected, Tab. 11.2 shows higher costs and lower availability than Tab. 11.1 due to a more spread out maintenance need. Using a prognostics tool and PMOPT in this scenario also yields significant results. Downtime can be reduced by 65 % for a schedule with no advantageous opportunities, compared to the current state of practice. In the case where seasonal opportunities are present, downtime can be reduced from 259 to 11.6 days.

11.6.2 Comparison with CPLEX

In order to investigate how far away from the optimum the results from PMOPT are, we formulated MSOP as a mixed integer linear programming problem. We

used ILOG CPLEX 9.0 on a mainframe computer with a 2.2 GHz Dual Core AMD Opteron CPU and 8 GB of RAM to solve the problem. The total runtime for each case was limited to 8 hours. Although running an algorithm for such a long time is not suitable for our needs, the comparison still gives us valuable insight in where PMOPT can be improved. In contrast, PMOPT was run on a laptop with a 1.6 GHz Intel CPU for 30 seconds in each case.

Results for the eight different cases (described previously) are compared in Tab. 11.3. *Diff* gives the relative difference between the best found cost for PMOPT and CPLEX, with negative values indicating that CPLEX found a better solution than PMOPT. The *Gap* column gives the relative optimality gap (distance to the relaxed optimum) as returned by CPLEX, with higher values indicating that the gap is larger. The gap is infinite if no feasible solution was found within 8 hours. The *Time* column report CPU runtime for proving optimality, with a cutoff at 8 hours.

For the two cases with a new turbine and seasonal stops, CPLEX was able to find the exact optimal solution (indicated by a gap value of 0). For the two cases with a used turbine and seasonal stops, CPLEX had found better solutions than PMOPT when 8 hours had passed, with a quite small optimality gap. While CPLEX produces slightly better results for cases with lifetimes from the prognostics tool (*Progn*), the instances with standard EOH lifetimes appears to benefit more significantly. It is notable that CPLEX reports a result which is more than 23% better than PMOPT in the case with EOH lifetimes and seasonal stops. However, when there are no seasonal stops, CPLEX cannot find a solution even close to the result from PMOPT within 8 hours.

11.7 Conclusions and Future Work

We described the development and deployment of an opportunity-based maintenance planning tool, PMOPT, specifically designed to fit the purpose of improving the maintenance schedules for gas turbines from SIT AB. The goal was to reduce both direct maintenance costs and production losses. Thanks to close collaboration with key personnel at SIT AB, we gained important insights into industrial maintenance planning, which allowed us to design and implement the maintenance planning tool. We believe that this has contributed greatly to the success of PMOPT.

We formally described and characterized the scheduling problem as NP-complete, and discussed a heuristic algorithm for solving it. Our experiments on a real-world example showed significantly reduced downtime (with up to 65%) and costs. Experiments with CPLEX gave even greater gains, but at the

cost of much longer solution times. Expected effects in practical use include large availability improvements, and preventive maintenance reductions with up to 12 %. Future plans include fleet level planning and labor resource optimization and scheduling, and application to other domains. We are also considering investigating solution sensitivity with regard to parameter changes.

Acknowledgment

We are thankful to Pontus Slottner and Per Almroth at Siemens Industrial Turbomachinery AB for providing expert knowledge regarding gas turbine lifetime predictions.

Bibliography

[1] M. BOHLIN, M. WÄRJA, A. HOLST, P. SLOTTNER, AND K. DOGANAY, *Optimization of Condition-Based Maintenance for Industrial Gas Turbines: Requirements and Results*, in *Proceedings of the ASME Turbo Expo*, 2009.

[2] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, MIT Press, Cambridge, MA, USA, second ed., 2001.

[3] R. DEKKER, *Applications of Maintenance Optimization Models: A Review and Analysis*, Reliability Engineering & System Safety, 51 (1996), pp. 229–240.

[4] J. E. G. COFFMAN, M. R. GAREY, AND D. S. JOHNSON, *Approximation Algorithms for Bin Packing: A Survey*, PWS Publishing Co., Boston, USA, 1997, ch. 2, pp. 46–93.

[5] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, January 1979.

[6] W. D. HARVEY AND M. L. GINSBERG, *Limited Discrepancy Search*, in Proceedings of the 14th International Joint Conference on Artificial Intelligence, 1995, pp. 607–615.

[7] O. LHOMME, *Consistency Techniques for Numeric CSPs*, in Proceedings of the 13th International Joint Conference on Artificial Intelligence, 1993, pp. 232–238.

[8] M. MARSEGUERRA, E. ZIO, AND L. PODOFILLINI, *Condition-Based Maintenance Optimization by Means of Genetic Algorithms and Monte Carlo Simulation*, Reliability Engineering & System Safety, 77 (2002), pp. 151–165.

[9] S. RUSSELL AND P. NORVIG, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Second ed., 2003.

[10] J. S. TAN AND M. A. KRAMER, *A General Framework for Preventive Maintenance Optimization in Chemical Process Operations*, Computers & Chemical Engineering, 21 (1997), pp. 1451–1469.

[11] R. E. WILDEMAN, R. DEKKER, AND A. C. J. M. SMIT, *A Dynamic Policy for Grouping Maintenance Activities*, European Journal of Operations Research, 99 (1997), pp. 530–551.

[12] Z. YAMAYEE, K. SIDENBLAD, AND M. YOSHIMURA, *A Computationally Efficient Optimal Maintenance Scheduling Method*, IEEE Transactions on Power Apparatus and Systems, PAS-102 (1983), pp. 330–338.

[13] O. C. ZIENKIEWICZ, R. TAYLOR, AND J. Z. ZHU, *The Finite Element Method*, Butterworth-Heinemann, sixth ed., 2005.

Glossary

A

A* search (A*) Best-first search procedure which uses an admissible heuristic estimate plus the path cost so far to decide which node to expand. *p. 29*

aperiodic task A task which is not released periodically. *p. 40*

availability The fraction of time a system is in an operational state. *p. 54*

B

best-case execution time (BCET) The lowest possible execution time of a task. *p. 48*

best-effort worst-case response-time The highest found response time for a task in a set of simulated or real measurements. *p. 13*

best-first search Search procedure in which the most promising node, as measured by a heuristic, is chosen for expansion. *p. 29*

blocking The prevention of a task to execute due to a shared resource being locked by another task. *p. 42*

branch and bound An algorithm for combinatorial optimization problems, where the candidates are systematically constructed. A hopefully large number of solution candidates are discarded by proving that all completions of a partial solution are non-optimal. This is done using upper and lower bounds on the objective function. *p. 26*

breadth-first search (BFS) Search procedure in which all branches of the current node are expanded before continuing with the next node, thus exploring the tree in a breadth-first manner. *p. 23*

C

chord An edge between non-consecutive vertices in a path. *p. 22*

chordal A graph is *chordal* if it contains no chordless cycles of length greater than three. *p. 22*

clique A complete subgraph. *p. 22*

combinatorial optimization problem A combinatorial optimization problem is an optimization problem where the set of feasible solutions is discrete. *p. 9*

condition-based maintenance (CBM) Maintenance which is based on more or less real-time condition data. *p. 51*

condition monitoring Continuous or non-continuous monitoring of an asset in order to determine its condition and operational status. *p. 55*

constraint A formal description of a requirement which must hold for all acceptable solutions to a problem. *p. 25*

constraint programming (CP) "A programming paradigm where relations between variables are stated in the form of constraints. Constraints differ from the common primitives of imperative programming languages in that they do not specify a step or sequence of steps to execute, but rather the properties of a solution to be found". From Wikipedia. *p. 27*

constraint satisfaction problem (CSP) The problem of whether a conjuction of constraints have a satisfying truth assignment. *p. 27*

controller-area network (CAN) A vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer. Communication is priority-based. See [64, 122, 176–180, 252]. *p. 39*

corrective maintenance (CM) Maintenance which is done in order to restore an asset to operational status after a breakdown. *p. 51*

D

deadline (DL) The latest time point a task should have finished its execution. *p. 40*

deadline monotonic (DM) A scheduling algorithm where tasks are prioritized according to increasing deadline. *p. 41*

depth-first search (DFS) Search procedure in which one branch of the current node is recursively expanded until a leaf is reached, thus exploring the tree in a depth-first manner. *p. 5*

directed acyclic graph (DAG) A directed graph which does not contain any cycles. *p. 1*

domain The set of possible values for a variable. *p. 27*

domain propagation The removal of all inconsistent values in a domain. . *p. 27*

E

earliest deadline first (EDF) A dynamic scheduling algorithm which always executes the active task with earliest deadline. *p. 41*

embedded system A computer system that is part of larger system, performing some of the functions of that system [120]. *p. 37*

Equivalent Operational Cycles (EOC) The total number of start-stop cycles of an industrial gas turbine. *p. 52*

Equivalent Operational Hours (EOH) A time unit used for industrial gas-turbine maintenance planning, based on the number of cycles, load, fuel quality, the presence of water injection, and the presence of significant exhaust temperature differences. *p. 52*

evolutionary algorithm A population-based metaheuristic optimization algorithm inspired by biological evolution. *p. 70*

execution stack Statically allocated memory, which is used during run-time to store return adresses, parameters in function calls and local variables. *p. 43*

execution time (ET) The time a task instance spends actively computing its result. *p. 39*

F

failure mode and effects analysis (FMEA) Analysis done to identify the different ways in which a machine can fail and the effects of such failure. *p. 262*

failure mode, effects and criticality analysis (FMECA) Extension of FMEA with a criticality analysis used to chart failure mode probabilities against the severity of their consequences. *p. 56*

first in, first out (FIFO) Processing of events according to a first-come first-served basis. *p. 45*

fixed priority pre-emptive scheduling (FPS) A scheduling algorithm where the active task with highest priority is always executed. *p. 41*

functional failure "The inability of an item to perform a specific function within specified limits", (from [171]). *p. 56*

G

genetic algorithm A specific type of evolutionary algorithm where individuals in the population are described by strings of data, and where individuals are reproduced using either combination of individuals, or mutation of single individuals. *p. 74*

H

hill-climbing algorithm An algorithm which explores a neighborhood of candidate solutions, selecting an improving candidate as the next step. The algorithm normally terminates when no improving neighbors can be found. *p. 32*

Hill-Climbing with Random Restarts (HCRR) An algorithm for provoking high response-times for a specific task in a real-time system. The algorithm is based on hill climbing and employing random restarts, for iteratively producing input to a real-time system. The generated consists of jitter, execution times, and external stimulus. *p. 1*

I

immediate inheritance protocol (IIP) A simplification of PCP where the priority of the locking task is immediately raised, even before other tasks are blocked on the same resource. *p. 42*

instance A specific task invocation. *p. 39*

interval graph A graph where the vertices can be represented by 1-dimensional intervals, and the edges correspond to interval intersection. *p. 23*

interval propagation The removal of values at the endpoints of the domain of a variable. *p. 27*

iterative improvement A search algorithm which iteratively applies improving transformations to a candidate solution. *p. 31*

J

jitter Time variation in a periodic event, such as the activation event of a task in a real-time system. *p. 46*

L

limited discrepancy search (LDS) Search method based on restricting search to paths which make at most a limited number of choices which do not agree with a given heuristic. LDS was introduced by Harvey and Ginsberg in [115]. *p. 29*

linear programming (LP) An optimization methodology in which the objective is a linear function, the constraints are linear inequalities, and variables are real-valued. *p. 25*

local search A search algorithm which only searches *locally* for improving solutions. *p. 31*

M

MABERA (MAB) An evolutionary algorithm for provoking high response-times for a specific task in a real-time system. The algorithm is based on mutation of chromosomes consisting of a random

number generator seed schedule, which in turn control the input generated to a program. See [141]. *p. 171*

maintenance policy A set of rules for when maintenance should be performed. *p. 53*

maximal clique (maxclique) A clique which is not a subgraph of a larger clique in the same graph. In other words, a clique is a maxclique if it cannot be extended with any node.

mixed integer programming (MIP) An optimization methodology in which the objective is to minimize (or maximize) a linear objective function, subject to a set of constraints expressed using linear inequalities, and where a subset of the variables are restricted to take integer values only. *p. 25*

multi-unit maintenance Maintenance where a system consists of multiple units or subsystems, and where the units interact in some way. *p. 64*

N

neighborhood A set of candidate solutions obtained from the current solution by local transformations in local search. *p. 32*

O

objective function A function which specifies what is to be optimized. Typical examples include utility or value (for maximization problems) and cost (for minimization problems). *p. 9*

offline scheduling A scheduling algorithm which follows a predetermined execution schedule. *p. 40*

offset The release time point of a task relative to the transaction activation time point. *p. 46*

online scheduling A scheduling algorithm which makes scheduling decisions during system execution based on the currently active set of tasks. *p. 40*

Opportunistic Maintenance (OM) Maintenance policy in which certain activities are performed prior to their optimal time point if this lowers the total cost of maintenance. *p. 65*

optimization Activity which aims at minimizing or maximizing an objective function. *p. 9*

P

Path Upper-Bound (PUB) An algorithm for computing the maximum stack usage in a transaction-based stack-sharing system, based on forming a preemption graph of possible preemptions and tasks, and, by using a longest-path algorithm, computing an upper bound on the maximal PPC. *p. 154*

perfect elimination order (PEO) An ordering of the vertices in a graph, such that each vertex forms a clique together with all adjacent vertices which occur later in the ordering. *p. 23*

periodic task A task which is periodically released for scheduling. *p. 40*

possible preemption chain (PPC) A sequence of task instances in increasing priority order, where each instance have the possibility of being preempted by all following instances. *p. 128*

Possible Preemption-Chain, Branch-and-Bound (PPCBB) An algorithm for computing the maximum stack usage in a transaction-based stack-sharing system, based on forming a preemption graph of possible preemptions and tasks, and, by using branch-and-bound, computing a maximal PPC. The bounding procedure used is PUB. *p. 155*

potential failure "A definable and detectable condition that indicates that a functional failure will occur", (from [171]). *p. 56*

precedence A relation between two tasks stating in which order they should execute. *p. 47*

predictive maintenance (PdM) Techniques and methods for predicting future breakdowns, so that preventive activities can be planned and performed in a timely fashion, thereby preventing the breakdown. *p. 55*

preemption chain A sequence of task instances $\upsilon_1, \upsilon_2, \upsilon_3, \ldots$ where υ_1 is preempted by υ_2, which is preempted by υ_3, and so on. *p. 125*

preemption threshold (PT) A priority threshold defined for each task, used to enforce that only tasks with a higher priority than the threshold are allowed to preempt the former task. *p. 71*

preventive maintenance (PM) Maintenance consisting of inspection, servicing and replacement tasks which are done in order to catch and prevent breakdowns from occurring. *p. 52*

Preventive Maintenance Optimizer (PM-opt) A software tool for planning maintenance, developed in Papers D–F. *p. 214*

priority ceiling protocol (PCP) A resource access protocol in which the priority of a task holding a resource is raised to a *priority ceiling* whenever another task is blocked on the same resource. *p. 42*

priority inheritance protocol (PIP) A resource access protocol in which the priority of a task holding a shared resource is raised to the same level as a higher-prioritized task trying to access the same resource. *p. 42*

prognostics and health management (PHM) Protective and diagnostic or prognostic devices and systems. Sometimes referred to as condition-based maintenance (CBM). *p. 59*

propositional satisfiability problem (SAT) The problem of whether a set of Boolean clauses on disjunctive normal form has a satisfying truth assignment. *p. 27*

R

random access memory (RAM) A type of volatile memory used for temporary storage during program execution. *p. 12*

rate monotonic (RM) A scheduling algorithm where tasks are prioritized according to increasing period time. *p. 41*

real-time operating system (RTOS) An operating system specifically engineered for real-time applications. *p. 39*

real-time system (RTS) A system in which timeliness is equally important for the system to work properly as is functional correctness. *p. 38*

release time The time instant that a task instance is released for scheduling. *p. 40*

reliability The fraction of time a system is either under preventive maintenance or in an operational state. *p. 54*

reliability-centered maintenance (RCM) Maintenance methodology and process in which a maintenance scheme based on the reliability of the system components is developed. *p. 56*

response time (RT) The time between the invocation of a task instance and the time point the instance finishes its execution. *p. 46*

response-time analysis (RTA) Family of techniques used to compute the response time of the tasks in a system under different scheduling policies. *p. 13*

RTSSim A real-time operating system simulator where tasks execute in a "sandbox" environment. The scheduling policy is preemptive priority-based scheduling. *p. 173*

S

schedulability analysis Design-time analysis aimed at determining whether a proposed real-time system is schedulable or not. *p. 41*

sporadic task A task whose releases have a specified minimum inter-arrival time. *p. 40*

Stack per Transaction Level Analysis (STLA) A method for computing an upper bound on the stack usage of a transaction-based stack-sharing system, in which the sum of the stack usage of each individual transaction, as computed by the PUB algorithm, is used. *p. 156*

stack resource policy (SRP) A mechanism for resource allocation which can eliminate deadlocks and unbounded priority inversion, invented by Baker in [24]. SRP will only allow jobs to enter the ready

queue when all of the resources they need are available. Also, in SRP a running task will inherit the priority of higher priority tasks blocked on any locked resource. *p. 44*

stack resource policy with preemption thresholds (SRPT) An algorithm for sharing resources in multiprocessor systems, together with a procedure for assigning preemption thresholds to tasks [93]. *p. 71*

Stack Upper Bound algorithm (PUB) An algorithm for computing an upper bound on the stack usage in a hybrid stack-sharing system, where the offsets and response times of tasks are used to compute a maximal PPC. *p. 129*

T

time-triggered protocol (TTP) An control system platform technology that supports the design of embedded systems. Communication is time-triggered. See [137] for more information. *p. 39*

transaction A group of tasks, each having an offset which is relative to a shared activation event. *p. 46*

transactional task model A task model in which tasks may have dependencies in their release times. *p. 46*

W

worst-case execution time (WCET) The highest possible execution time of a task. *p. 40*

worst-case response time (WCRT) The highest possible response time for a task. *p. 13*