Quality and Utility

On the Use of Time-Value Functions to Integrate Quality and Timeliness Flexible Aspects in a Dynamic Real-Time Scheduling Environment

Thesis

in partial fulfillment of the requirements of the degree of

Doctor of Science (Dr. rer. nat.)

at the

Faculty of Mathematics and Informatics

of the

University of Passau

Thomas Schwarzfischer

October 2004

Day of Oral Exam: 21 January 2005

Abstract

Scheduling methodologies for real-time applications have been of keen interest to diverse research communities for several decades. Depending on the application area, algorithms have been developed that are tailored to specific requirements with respect to both the individual components of which an application is made up and the computational platform on which it is to be executed. Many real-time scheduling algorithms base their decisions solely or partly on timing constraints expressed by deadlines which must be met even under worst-case conditions. The increasing complexity of computing hardware means that worst-case execution time analysis becomes increasingly pessimistic. Scheduling hard real-time computations according to their worst-case execution times (which is common practice) will thus result, on average, in an increasing amount of spare capacity. The main goal of flexible real-time scheduling is to exploit this otherwise wasted capacity. Flexible scheduling schemes have been proposed to increase the ability of a real-time system to adapt to changing requirements and nondeterminism in the application behaviour. These models can be categorised as those whose source of flexibility is the quality of computations and those which are flexible regarding their timing constraints. This work describes a novel model which allows to specify both flexible timing constraints and quality profiles for an application. Furthermore, it demonstrates the applicability of this specification method to real-world examples and suggests a set of feasible scheduling algorithms for the proposed problem class.

Zusammenfassung

Einplanungsverfahren für Echtzeitanwendungen stehen seit Jahrzehnten im Interesse verschiedener Forschungsgruppen. Abhängig vom Anwendungsgebiet wurden Algorithmen entwickelt, welche an die spezifischen Anforderungen sowohl hinsichtlich der einzelnen Komponenten, aus welchen eine Anwendung besteht, als auch an die Rechnerplattform, auf der diese ausgeführt werden sollen, angepasst sind. Viele Echtzeit-Einplanungsverfahren gründen ihre Entscheidungen ausschließlich oder teilweise auf Zeitbedingungen, welche auch bei Auftreten maximaler Ausführungszeiten eingehalten werden müssen. Die zunehmende Komplexität von Rechner-Hardware bedeutet, dass die Worst-Case-Analyse in steigendem Maße pessimistisch wird. Die Einplanung harter Echtzeit-Berechnungen anhand ihrer maximalen Ausführungszeiten (was die gängige Praxis darstellt) resultiert daher im Regelfall in einer frei verfügbaren Rechenkapazität in steigender Höhe. Das Hauptziel flexibler Echtzeit-Einplanungsverfahren ist es, diese ansonsten verschwendete Kapazität auszunutzen. Flexible Einplanungsverfahren wurden vorgeschlagen, welche die Fähigkeit eines Echtzeitsystems erhöhen, sich an veränderte Anforderungen und Nichtdeterminismus im Verhalten der Anwendung anzupassen. Diese Modelle können unterteilt werden in solche, deren Quelle der Flexibilität die Qualität der Berechnungen ist, und jene, welche flexibel hinsichtlich ihrer Zeitbedingungen sind. Diese Arbeit beschreibt ein neuartiges Modell, welches es erlaubt, sowohl flexible Zeitbedingungen als auch Qualitätsprofile für eine Anwendung anzugeben. Außerdem demonstriert sie die Anwendbarkeit dieser Spezifikationsmethode auf reale Beispiele und schlägt eine Reihe von Einplanungsalgorithmen für die vorgestellte Problemklasse vor.

Acknowledgements

The best way to become acquainted with a subject is to write a book about it.
Benjamin Disraeli

Gratitude is merely the secret hope of further favours.
François de La Rochefoucauld

As is the case in most circumstances in life, achievements are not normally possible without the assistance of numerous well-meaning people. The same was true for the undertaking of carrying out my PhD research.

Representing so many who contributed to the successful completion of this work either through active help or by mental support, I would like to thank the following persons by name. I thank Prof. Dr.-Ing. Werner Grass for his long lasting assistance and guidance, numerous impulses to gain new insights and especially for helping me place my work into a bigger context. I am grateful to Prof. Dr.-Ing. Jürgen Teich of Friedrich-Alexander University in Erlangen for acting as an external referee, and to Prof. Dr. Gottlieb Leha, Prof. Christian Lengauer, PhD, and Prof. Dr. Volker Weispfenning to sit on the committee for the oral examination. The advice of Prof. Dr. Niels Schwartz was essential for coming to terms with formal requirements of the graduation process. I say my thanks to all the people in our group, especially to Dr. Bernhard Sick for his cooperation, Martin Grajcar for various spontaneous ideas and Markus Ramsauer for numerous fruitful discussions while working on the same project. I extend my thanks to Franz Rautmann for keeping our technical equipment running most of the time and Eva Kapfer for her constant effort to establish a hearty and welcoming climate at the institute.

I thank our students for their participation in the PaSchA project: Christian Jünger, Klaus Ehrnböck, Jakob Flierl, Kerstin Meyerhofer, Selçuk Demirci, Robert Schiller, Katrin Limpöck, Diana Lucic, Matthias Weindler, Peter Zach, Bettina Riedmann, Barbara Busse, Christian Müller, Ralf Zimmermann and others I might have forgotten to mention.

I also greatly appreciated the comments of Prof. Dr. Hiroshi Nagamochi, Dr. Koji Nonobe and Dr. Mutsunori Yagiura of the University of Kyoto during the final stages of my PhD project, when I could spend interesting months at their institute.

I thank all those who encouraged me to carry on with endless patience every time I had lost the necessary confidence, first of all Michiko Araseki for all her love. Finally, I am deeply indebted to my parents, Agnes and Karl Schwarzfischer, who gave me the opportunity to pursue my studies, and to God for providing me with the abilities to do so.

Passau, January 2005

Thomas Schwarzfischer

Contents

List of Figures

List of Definitions

Chapter 1

Introduction

Begin at the beginning and go on till you come to the end; then stop.
Lewis Carroll

Deciding on appropriate actions within a system to achieve pre-defined goals, to meet certain given constraints or as a reaction to the behaviour of the environment is a very general description of problems existing in a wide variety of application areas, ranging from the field of economics to computer science and engineering disciplines. Several terms have been coined for a series of similar basic problems including planning, scheduling, allocation, timetabling, and configuration. Most prominently, the expressions planning and scheduling are both frequently used for concepts linking a set of environmental parameters to a methodology deciding on which actions to perform at which time, in which order and using which resources. Traditionally, a planning problem is defined on sets of states (including an initial state), actions, and one or several goals which are to be achieved. The aim of a planning agent is to select appropriate actions in each state and to perform them in a suitable order so that the system state ultimately transits from the initial to (one of) the goal state(s). A typical planning problem is travel planning, where the result is an itinerary given a point of departure and a destination. On the other hand, traditional scheduling problems are cast in terms of a set of activities and several kinds of constraints (e.g., resource, timing, or precedence constraints). The scheduler has to ensure that resources are allocated to activities appropriately at each time instant, so that the constraints can be met. Maybe the best-known classical scheduling problem is job-shop scheduling, where sets of tasks must be executed on a set of machines subject to the precedence constraints that may have been posed on the task set. Although the terminology within this work was taken from the scheduling area of research, the close relationship of the specific class of scheduling problems to planning means that the results are applicable also to problems of the planning domain.

1.1 The Relationship between Planning and Scheduling

Both classes of problems can be categorised in several ways. Dean and Kambhampati [DK96] suggest the following:

1. aim: either find some solution satisfying the constraints and being sufficiently close to a goal state (*satisficing*) or find the least cost or most rewarding solution satisfying the constraints (*optimising*)
2. the a-priori knowledge of the dynamics of the environment: either *deterministic*, *nondeterministic*, or *stochastic*
3. representation of plans or schedules: either *conditional* (depending on future behaviour of the environment) or *unconditional* (independent from future behaviour of the environment)
4. time variance of plans or schedules: either *stationary* (depending on current time) or *time variant* (independent from current time)
5. adaptivity: either *closed-loop* (consequences of prior actions influencing future actions) or *open-loop* setting (consequences of prior actions not influencing future actions)
6. performance measure: *goal-based* (distance to final state) or *cost-based/reward-based* (discounted cumulative cost or reward)
7. deliberation time: *off-line* (planning / scheduling prior to execution) or *on-line* (planning / scheduling concurrent with execution)

Despite the similarities in the problem settings and the methods applied to solve them, there is a common agreement that planning focuses more on action selection and action ordering, whereas scheduling is more concerned with resource assignment and exact timing issues [Sau03]. It has been noted, however, that most practical applications feature characteristics from both areas. Contrary to the basic layout of planning problems, real-world applications tend to require the ability to handle metric quantities, overlapping actions with finite durations, and very often some notion of a resource model. Similarly, most real-world scheduling problems require more than only allocating resources to pre-specified activities over time; it is frequently necessary to solve subproblems with planning characteristics like a simple selection among alternative processes (e.g., due to heterogeneous resource capabilities) or the synthesis of complex networks of activities. We opted for the term *scheduling* in order to emphasise on the resource allocation and real-time aspects in our specific problem, rather than the precedence ordering and alternative selection aspects. Nevertheless, our model is also applicable to related time-aware planning domains.

In the following sections we will outline the motivation of our work and classify the approach according to the criteria of [DK96].

1.2 Basic Real-Time Scheduling Terminology

Generally speaking, scheduling problems deal primarily with allocating resources to activities over time. Resources can be machines in a production plant, personnel within a company, vehicles of a transport enterprise, or processors of a computer system. The common terminology identifies some of the resources needed to make progress in the problem domain as the major driving entities and refers to them as *processing units*. Scheduling domains differ largely in the order of magnitude of the time ranges within which they have to take action. Whereas many production or personnel scheduling problems are aimed at time ranges of days, weeks, or even months, processor scheduling usually deals with times in the area of milliseconds and below. Obviously, the vast differences in the time domains result in different techniques to be applicable, even though the basic problem aspects are similar. For the remainder of this thesis, we concentrate on scheduling problems for computing machinery, i.e., with allocating processors to the elements of an application program.

All of today's widely used operating systems support multitasking, which allows multiple computations to run concurrently, taking turns using the processing units and other resources of the computer. This has made it necessary to come up with elaborate schemes for distributing these shared resources among different computations; processor scheduling deals with exactly the problem of finding suitable distribution schemes.

In computer science, an *application* is the use of a technology, system, or product. More specifically, the term application is a shorter form of *application program*, i.e., a program designed to perform a specific function directly for the user or for another program. Applications use the services of the computer's operating system and other supporting applications and are organised internally as sets of smaller units called *tasks*. In real-time scheduling, a task is a basic unit of programming that the operating system or the runtime environment controls. Note that unfortunately, the expressions *task*, *process*, *activity*, or *job* are frequently used interchangeably; however, these terms do have distinct meanings in different contexts and may be used to add some structure to a task set (e.g., an application consisting of jobs, which in turn consist of tasks). The time when a task becomes known to the system is called its *release time*; within this thesis, we assume that the release time equals the earliest time of activation of a task. Obviously, the processing units in this class of scheduling problems can be identified as the set of general-purpose and application-specific processors, and further resources are, e.g., memory units, busses, and peripheral devices.

We are not going to investigate into methods for solving scheduling problems for applications without explicit timing constraints like office software, web browsers, or drawing programs; scheduling schemes for such problems are incorporated into any general-purpose operating system. In most cases, their objective is to maximise the throughput or to minimise the average waiting time. Instead, we assume applications to be executed under real-time conditions. A common misconception is that real-time operation is synonymous to fast operation. Real-time computation can better be defined as the ability of a system to guarantee the completion of operations within given time limits, not only under average-case, but also under worst-case conditions. Unlike traditional (non-real-time) computing systems defining correctness solely as operational correctness, i.e., as compliance with a given correlation of out- and input, real-time systems must respond in a (timely) predictable way to possibly unpredictable external events. In other words, *correctness* in real-time systems consists of both *operational* and *temporal correctness*. Looking at the response time of computations, we notice that for a non-real-time application like a word processing program, a small *average* reaction time to user input is desirable, but quite long delays in rare cases are acceptable. On the other hand, the same behaviour is clearly not acceptable for the fly-by-wire system of an aircraft: a guaranteed *maximum* response time for all safety-critical tasks under all possible circumstances is absolutely essential.

Every real-time processor scheduling system includes two basic components: a *scheduler* determining which resources to allocate to which task at which time, and a *dispatcher* responsible for actually allocating and revoking the processor based on the results of the calculations of the scheduler. The method used to derive scheduling decisions from the state of the application and its environment is called the *scheduling algorithm*. Scheduling algorithms are frequently classified into *static* (*offline*) and *dynamic* (*online*) ones. Static scheduling algorithms have the advantage that they do not incur any significant overhead at runtime, but on the other hand they can obviously not react appropriately to changing characteristics of the application, so that they are only suitable for problems with well-known deterministic behaviour with regard to release times, execution times, etc.

Undoubtedly, for a certain class of safety and timeliness critical applications, worst-case analysis of task execution times and specifying timing constraints as deadlines which must always be guaranteed to be met are justified. On the other hand, a different kind of applications has gained attention in recent years; these applications do have properties related to the timing of computations, but exhibit a higher degree of flexibility than the traditional real-time scheduling approaches. The major advantage of these so-called *flexible scheduling* schemes is to avoid the pessimistic worst-case analysis, which typically leaves the (presumably expensive) hardware of the computer unused in the average case. Unfortunately, no consistent terminology has been agreed upon so far. However, attempts have been made to classify techniques that allow trading off at runtime properties of the results of a computation and the effort in terms of time and resources these computations need to produce the results. These classifications refer to the *load* of the system. This expression has two different meanings. The first one is the percentage of time within a given interval that the processing units are busy executing tasks. Obviously this *processor load* ranges between 0 and 1. On the other hand, the *application load* (also called *utilisation*) denotes the processor load that would result from all tasks being executed and completed; of course, the application load can exceed 1. A system is called *overloaded* if the application load is higher than 1. A desired property of scheduling algorithms is graceful degradation under overload conditions, which means that the performance does not drop dramatically, but gradually beyond the limit of an application load of 1. For our class of scheduling problems, there are two parameters which the scheduler can compromise on to allow the application to degrade gracefully when in overload. Liu [Liu00] divides flexible scheduling techniques into two broad categories, depending on whether they are designed for graceful degradation in result quality or in timeliness. For brevity, we will use the expression *utilisation* for application load and the expression *load* for processor load.

1.3 The Quality of Computations

The first source of flexibility in scheduling models we consider arises from giving up the so-called run-to-completion assumption.

1.3.1 The Run-to-Completion Assumption

Traditional scheduling schemes almost always rely on a concept called the *run-to-completion assumption*. This is to say that a scheduler has no means of influence on the execution times of tasks other than the selection of the processing unit on which it places them to execute and the allocation of a certain amount of computation time on these processing units. The notion of a task finishing successfully lies entirely within the task itself. This assumption does make sense in a broad variety of application scenarios. For example, a scheduler for the disc memory of a computer certainly does know the order in which it wishes to serve the individual requests, and it can estimate their start and end times. Depending on the nature of the specific scheduling algorithm and the dynamics of the requests, these estimates may be more or less accurate. Even though the scheduler is well-informed about the environment it has to work in (which in many real-world applications is not the case either), it does not have the ability to decide on the duration of activities to any extent. The data on individual activities available to the scheduler can be, e.g., fixed execution times or probability distributions for execution times. Figure 1.1 shows the general model of a scheduler for tasks in the run-to-completion category.

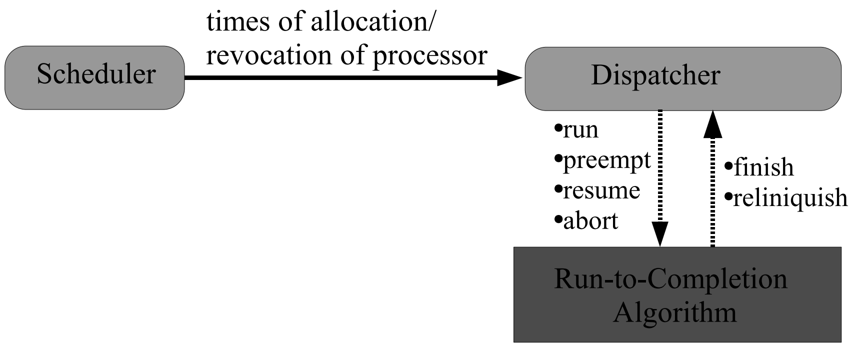


Figure 1.1: Run-to-completion scheduler model

The dispatcher can

- cause the task to *run* on the processor, i.e., to let it start computing
- *preempt* the execution of the task, so that it can later be resumed
- *resume* the execution after prior preemption or relinquishment
- *abort* the execution, i.e., terminate its execution without reasonable result

The run-to-completion task can

- *relinquish* the processor voluntarily, so that it can later be resumed
- *signal* the successful termination of the computation

Note that not all of these actions are possible for every scheduler. Typically not more than one of the actions “relinquish” and “preempt” is defined, the former one (or neither of them) for non-preemptive, the latter one for preemptive scheduling algorithms.

1.3.2 Quality-Flexible Scheduling

In other circumstances, however, the scheduler has an additional degree of freedom inasmuch as the execution times can be modified to increase or decrease the quality of the computations of the tasks. Several possibilities exist for quality-flexible scheduling:

- In some cases, there may be several implementations for a specific problem available which differ in both the duration and their quality or accuracy.
- Tasks may be implemented as iterative algorithms under the assumption that rising allocation of computation time to an activity results in higher quality of this activity and hence in a higher contribution to the overall performance.
- Tasks may be parameterised prior to execution such that their duration can be fine-tuned and adapted to the resources currently available in the system.

Several models on these aspects of computation have been described, and we will go into much more detail on these in a later chapter. However, two basic schemes for quality-flexible scheduling can be identified.

In the first one (figure 1.2), parameters are passed on from the scheduler to the application tasks via the dispatcher to adapt their level of service (i.e., their quality) appropriately to match the resources available.

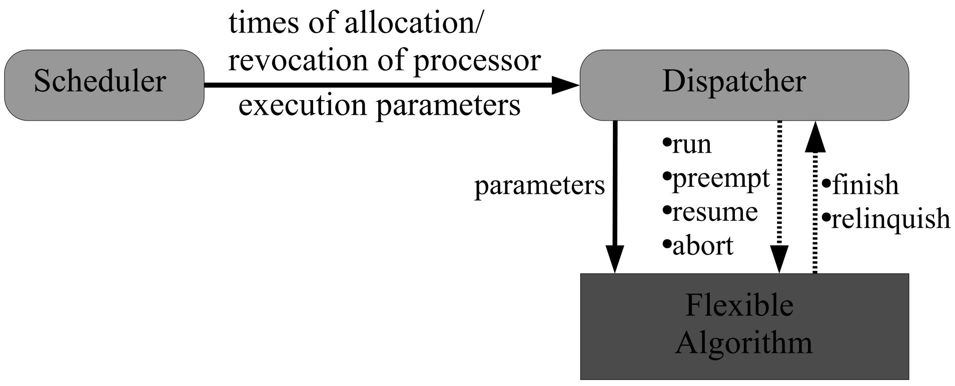


Figure 1.2: Quality-flexible scheduler model, parametrisation type

In the second case (figure 1.3), application tasks are directly terminated by the dispatcher. In the latter model, there is no notion of unsuccessful computations (at least not resulting from timing constraints); there is no edge indicating the abortion of a task and the “finish” action is signalled from the dispatcher to the task, not the other way around.

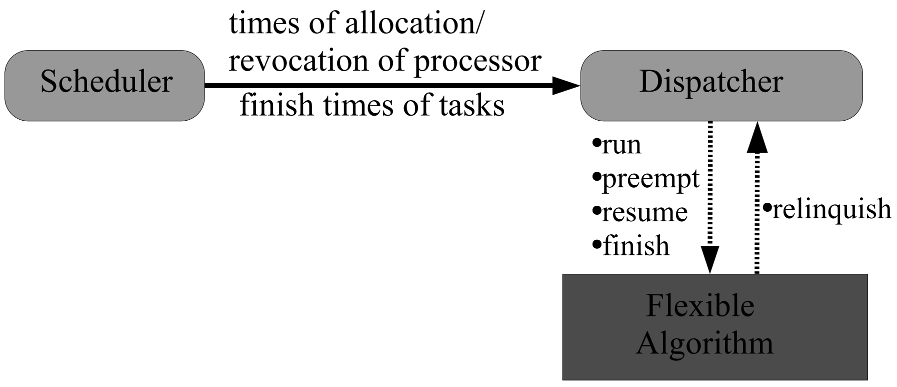


Figure 1.3: Quality-flexible scheduler model, external termination type

Quality functions describe the relationship between the execution time awarded to a task and the quality to be expected from this task. Figure 1.4 shows an example quality function.

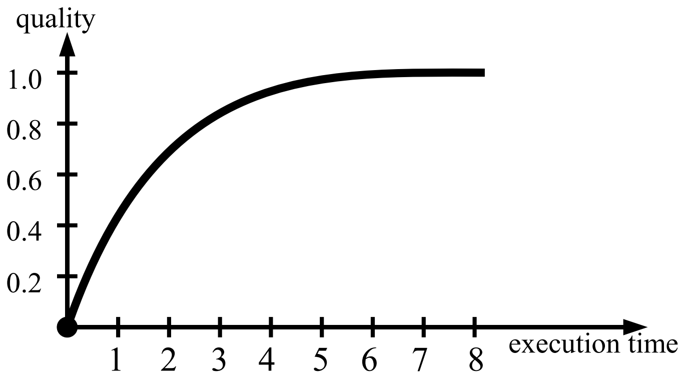


Figure 1.4: Example quality function

1.4 The Timeliness of Computations

The second source of flexibility we consider is based on timing constraints less strict than traditional deadlines.

1.4.1 Traditional Timing Constraints in Scheduling

As stated above, of all the parameters that are used to drive the scheduling process, timing constraints are among the most common ones. The simplest way of specifying timing constraints on individual tasks is by means of deadlines by which their execution is supposed to have finished. Depending on the consequences of a missed deadline to the performance of the overall application, attributes are usually assigned to a deadline, e.g., hard, soft, semi-hard, firm, etc. Deadlines can be specified relative to the release time of individual activities or in terms of a system-global time.

1.4.2 Timeliness-Flexible Scheduling

Alternatives to specifying deadlines are end-to-end constraints, where in general maximum delay values are given for entire chains of tasks rather than for individual tasks. Furthermore, so-called window constraints can be posed on tasks that enter the system repeatedly on a more or less regular basis. Window constraints require a certain percentage within any consecutive number of tasks to meet their deadlines. A direct generalisation to specifying timeliness with traditional deadlines is by functions of time, so that levels of urgency can be modelled in a much more fine-granular way; we use the term *utility functions* for the mentioned functions of time. Figure 1.5 shows an example utility function.

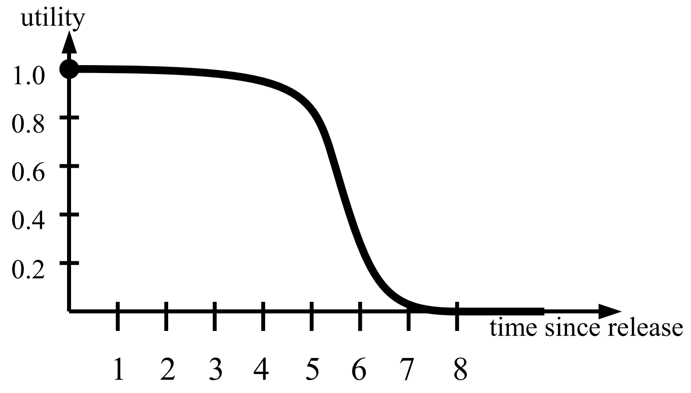


Figure 1.5: Example utility function

1.5 State of the Art

Until recently, the paradigms of scheduling imprecise computations (quality-flexible scheduling) and scheduling with timing constraints given as functions of time (timeliness-flexible scheduling) used to be considered orthogonal. Handling imprecise computations has been more popular in the planning than in the real-time scheduling research community. Rare attempts have been made to add deadlines to anytime planning schemes, and looking at the few existing approaches to timeliness-flexible scheduling, no evidence could be found on attempts to do without the run-to-completion assumption. However, it was recognised that there are certain environments where components of an application can be described very naturally as imprecise computations (e.g., multimedia applications, image processing, etc.) and the context of the application shows properties that can be modelled better with fine-granular utility functions than deadlines. The general assumption, however, was that the expected complexity of the search for solutions to any such problem was too high for the approach to be of any practical use [BPB$^+$00]. Table 1.1 shows a first classification of scheduling schemes along the two categories with which this work primarily deals, namely the sources of flexibility. It is especially the class of quality-flexible and timeliness-flexible schemes (which we call *quality-utility scheduling* schemes) we are going to investigate.

	Deadlines	Flexible Timing Constraints
Run-to-completion tasks	traditional real-time scheduling schemes	timeliness-flexible scheduling schemes
Quality-based schemes	quality-flexible scheduling schemes	quality-utility scheduling schemes

Table 1.1: Classification of scheduling schemes

Note that not all real-time scheduling schemes can be classified according to this diagram. For example, not all scheduling schemes (e.g., rate-monotonic scheduling and many other static-priority algorithms) make explicit use of the timing constraints to drive the scheduling decisions.

1.6 Example

As a first (simplistic) example for the quality/utility scheduling problem consider the scenario of a multimedia application which generates three-dimensional scenes of moving objects in real-time, i.e., concurrently with the display of these scenes. Assume that due to user interaction (changing the viewing perspective) the parameters to calculate the scenes are not available before runtime. The objects can be displayed in various granularities, where a subdivision scheme allows to generate one image from the previous one. Obviously, the granularity (and hence the image quality) rises with increasing computation time. The example of figure 1.6 shows a sphere-shaped object in rising granularity levels, starting from a cube.

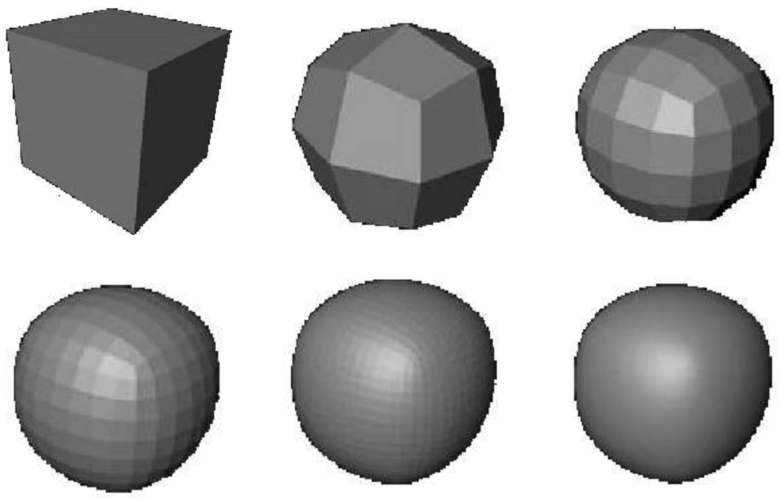


Figure 1.6: Iterative refinement of object

Next, look at the snapshots of a sequence of moving and rotating objects of figure 1.7. To let the sequence appear smooth to the human eye, scenes have to be generated at roughly equidistant times, and the distance must not be too big. However, minor deviations from either the periodicity or a desired maximum temporal distance between frames may be acceptable for the sake of the quality individual objects are rendered with.

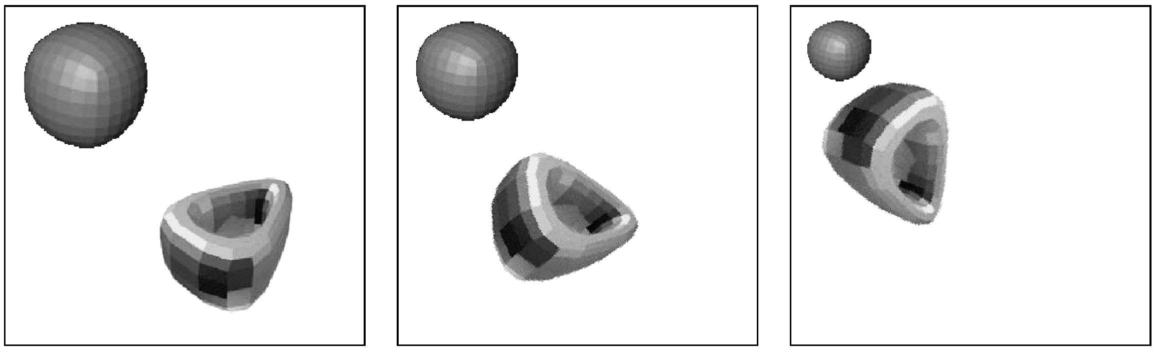


Figure 1.7: Snapshots of sequence of moving objects

Obviously, there is a tradeoff between the quality of image rendering and the timeliness of such a procedure. The functions of figures 1.4 and 1.5 can serve as the quality and utility functions of the rendering tasks.

This kind of problem is usually treated by keeping either the quality specifications fixed or providing hard deadlines for the rendering algorithm. In the former case, the execution time of the rendering algorithm can only be roughly estimated, and the influence of other applications running concurrently on the same processor is not normally taken into account directly; the best one can hope for is that the processes producing the image stream are prioritised sufficiently high to allow for a more or less acceptable display frequency. The latter choice tries to guarantee the timely generation of scenes; however, such guarantees can only be given by worst-case execution time analysis, so that the quality parameters will generally be set at a level which is unnecessarily low in the average case. No previous model for scheduling problems directly addresses the tradeoff between quality and timeliness for other than very special cases.

1.7 Aims of this Work

This work intends to show that there is indeed a way to specify scheduling problems with quality profiles and fine-granular timing constraints associated with the individual tasks of an application. In the terminology cited earlier in this chapter, the system model we develop is dynamic with stochastic knowledge of the scheduler on its environment, the scheduling problem is cast in terms of optimising an objective function defined as the cumulative reward of the tasks. In our model, an application is structured hierarchically in a *task / subtask relation*. Logical types (either *and* or *or*) are associated to the nodes of the task hierarchy, so that we can interpret tasks differently in their role within the hierarchy: *or* type tasks represent the case where the children are alternative implementations of the parent node, and *and* type tasks mean that the parent node is composed of the child nodes. The task / subtask model is more general in expressiveness than the hierarchisation by jobs and tasks mentioned earlier, as it can span several levels.

The model of a generic example application consisting of tasks with release types and logical type specifications (circles in upper part) with hierarchy graph (continuous lines) and dependency edges (continuous arrows) is shown in figure 1.8. The bottom part of the model is made up of methods (rectangles) and processors (ellipses) with the associated access edges.

Another important characteristic of our model is that it deals with task sets which vary over time; for this reason, all scheduling algorithms we develop are bound to work dynamically, as the decisions depend on the system state unknown prior to starting the application. Partial conditional or unconditional schedules are calculated repeatedly according to an up-to-date estimate of the future behaviour of the environment; these schedules are generally valid only relative to a specific time and thus time-variant. In addition to the hierarchy relation, a precedence relation is introduced as a second graph structure on the same task set. The hardware platform consists of a heterogeneous multiprocessor system with shared memory, but no other resources. To facilitate the reuse of basic algorithms, these are provided as a library of so-called *methods*. Task

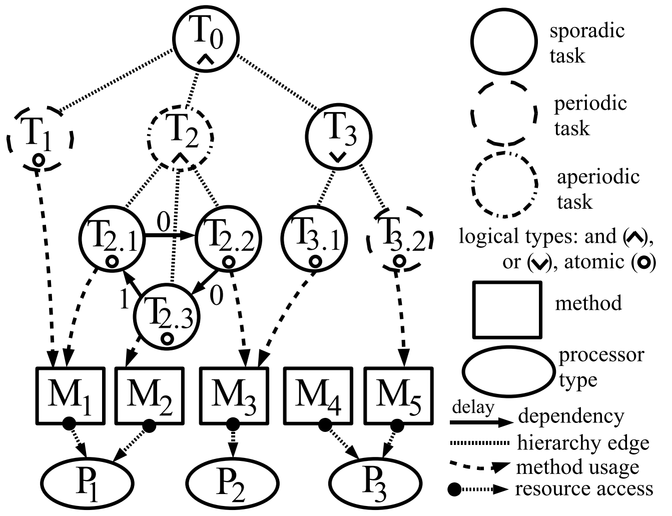


Figure 1.8: Example application graph

hierarchies are built upon this method layer.

We also investigate a feedback mechanism in order to allow the scheduler to adapt its own parameters while monitoring the environment and the consequences of its own actions. Furthermore, we will propose scheduling algorithms that solve practical scheduling problems of this class.

1.8 Structure of the Thesis

After the introduction of a basic model for quality / utility scheduling of unstructured task sets on a single-processor architecture, several scheduling algorithms are described to work on this problem class. In the subsequent chapter, the initial model is extended to hierarchical task networks for the description of an application and to heterogeneous multiprocessor architectures as the target platform. After that, we present a control-theoretic approach to closed-loop quality / utility scheduling. Descriptions of real-world examples in the realm of this methodology and of the simulation environment developed in order to evaluate (amongst others) the performance of the scheduling algorithms introduced in this work are followed by a series of empirical results. The thesis concludes with a detailed picture of the scientific context in which this work can be seen.

1.9 Publications

Parts of the ideas presented in this thesis have previously been published: [Sch03c] introduced the basic idea of investigating quality and timeliness aspects of applications. [Sch03b] extended the model to aperiodic task sets and instance graphs, and [Sch03d] described a detailed value-based representation of precedence constraints. Finally, [Sch03a] presented extended simulation results for a scheduling scheme based on local search, and [Sch04b] an alternative decision-theoretic scheduling scheme and a control-theoretic feedback mechanism.

Chapter 2

The Basic Quality / Utility Scheduling Problem

A cynic is a man who knows the price of everything, and the value of nothing.
Oscar Wilde

In this chapter we describe the basic quality / utility scheduling problem. We introduce a preliminary simplified version of the system model sufficient for the discussion of the scheduling algorithms in the following chapter. This model will be extended later in this thesis.

2.1 Basic Quality / Utility Scheduling Problem

As we want to emphasise the quality and timeliness flexible aspects, we restrict our attention in this chapter to non-hierarchical task sets without precedence constraints. Furthermore, we only consider single-processor systems as the hardware platform for the time being.

2.1.1 Application Model

As stated earlier, an application in our model consists of a (possibly infinite) set of tasks:

Definition 1 (Task set)
We denote the set of all tasks of the application by $\mathbb{T} := \{T_1, T_2, \ldots\}$.

The main attributes of tasks are the quality and utility functions. Quality and utility functions are defined on different time domains. As utility functions express the urgency of computations and urgency is independent of these computations actually taking place or not, they are defined in terms of a system-global time. On the other hand, the quality of a task depends on the amount of computation time that has been or will be awarded to the task; hence, quality functions are defined in terms of a task-local time. A schedule for a single-processor system can be fully represented by a series of functions mapping global time to processor time allocated to each task up to this global time. An example series of partial schedules is shown in figure 2.1. The global clock advances with each time instant, the local time of a task only if the task is scheduled for execution.

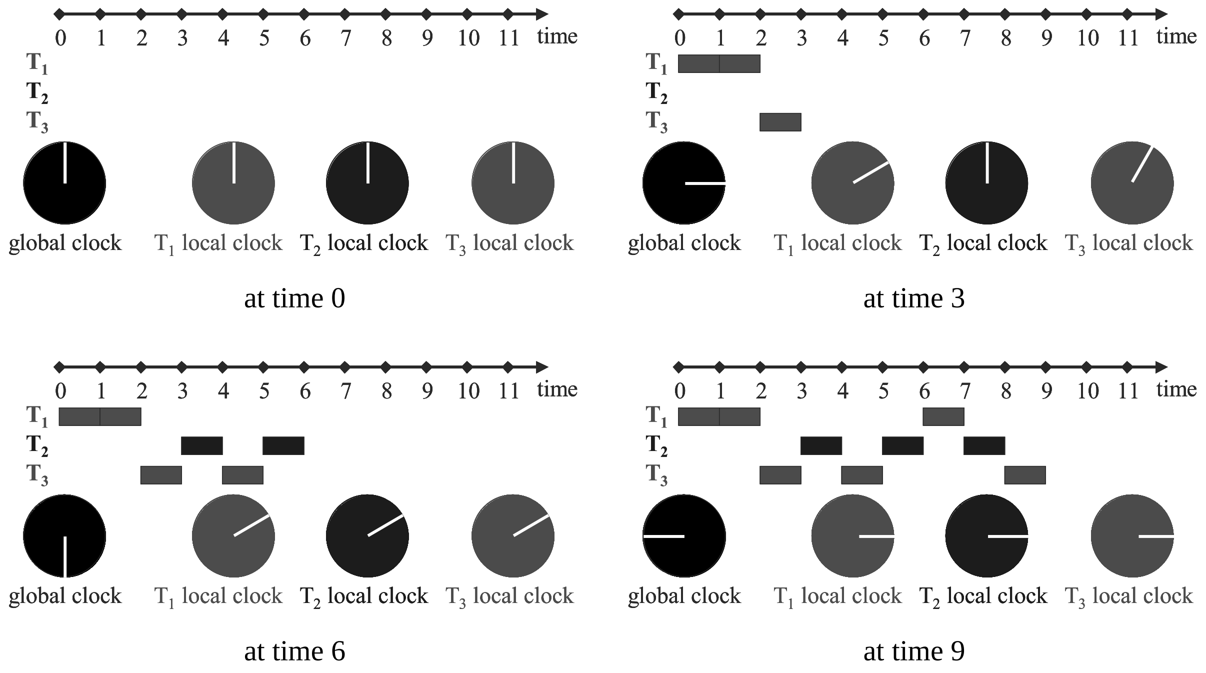


Figure 2.1: Partial schedules, local and global times

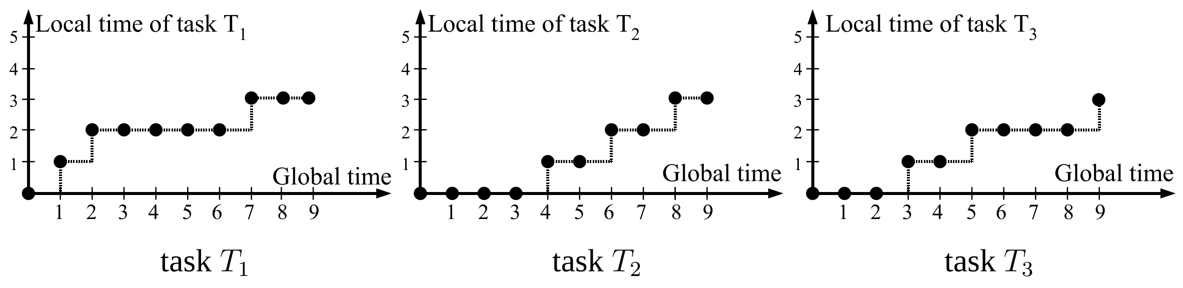


Figure 2.2: Functions mapping global time to local times of tasks

This schedule translates into three functions mapping global time to local time for each task (figure 2.2).

As our time model is discrete, global times are simply natural numbers; however, for the sake of clarity in later definitions, we introduce a new symbol for the set of global time instants.

Definition 2 (Global time)
The set of global time instants is denoted by $\mathbb{GT} :\equiv \mathbb{N}_0$.

The release times of tasks are the times when the environment and the scheduler gain knowledge of their existence; in our model, this time equals the time of earliest activation of a task. Although the system and especially the scheduler do not know in advance the exact release times of tasks, we assume that stochastic distributions of release times are available. However, offline scheduling is clearly unsuitable, and online scheduling schemes have to be developed. We do not want to employ specialised hardware to schedule applications, but instead use the same processor for both scheduler and application tasks. These preconditions impose very hard restrictions on the complexity of scheduling algorithms, and we will very likely have to resort to fast heuristics instead of optimal search algorithms. Furthermore, our task model is preemptive, i.e., we assume that tasks can be interrupted and resumed at any time and as often as necessary. Context switch costs are not taken into account.

Definition 3 (Release times)
The release time of task $T \in \mathbb{T}$ *is* $r_T \in \mathbb{GT}$.

Just like global times, task-local times are discrete; however, an additional symbol is introduced for clarity. The task-local time of a task indicates the amount of processor service awarded to it. We say that a task has reached local time n at (global) time t if it was allowed to run on the processor for n time units until time t.

Definition 4 (Local time)
The set of local time instants of task $T \in \mathbb{T}$ *is* $\mathbb{LT}_T :\equiv \mathbb{N}_0$.

The task set $\mathbb{T}$ encompasses all tasks that have been or will ever be released. The dynamic scheduler, however, works on a subset of tasks most relevant at the time of making scheduling decisions. This subset of interest consists of tasks already released in the past and tasks likely to be released in the near future. Obviously, this subset (which we will subsequently denote by $\mathbb{T}' \subseteq \mathbb{T}$ in many cases) has to be adapted regularly. In principle, tasks could remain in the set of interest forever after their release time; however, for efficiency reasons, a scheduler should remove tasks from the set once they cannot contribute much to the overall performance any more.

2.1.2 Quality and Utility Functions

Quality and utility functions are defined for each task $T \in \mathbb{T}$. The quality function is based on task-local time:

Definition 5 (Quality functions)
A quality function q_T for task $T \in \mathbb{T}$ is defined as a monotonically increasing function

$$q_T : \mathbb{LT}_T \to \mathbb{R}_0^+.$$

Quality functions $q_T(n)$ have bounded values for $n \to \infty$.

Quality functions are evaluated according to the computational progress of a task; after having been allowed to run on the processor for n time units, the quality of the task T is $q_T(n)$.

Examples for quality functions can be seen in figure 2.3.

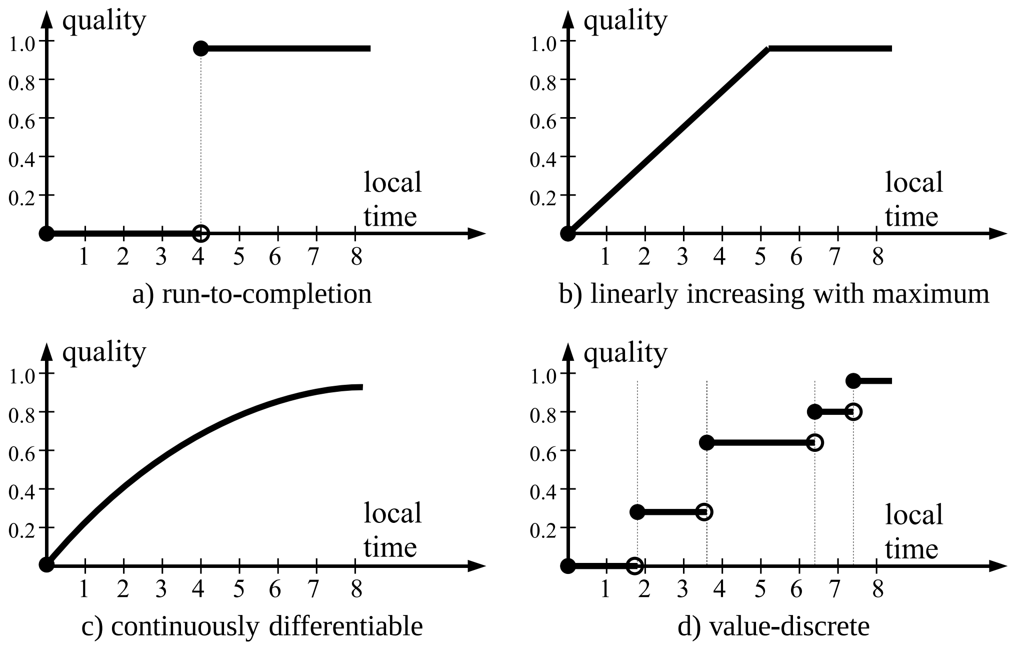


Figure 2.3: Example quality functions

Example a) models a run-to-completion task, which accrues value only if it reaches its execution time; no additional reward is gained after that. Example b) has a quality linearly increasing with execution time up to a given maximum value; this maximum is approached asymptotically by the continuously differentiable, concave function in c). The function in example d) is value-discrete.

Note, however, that in our model time domains are discrete and hence, quality functions can only be approximations of the continuous example functions. Note also that the quality function of a method need in fact not be invariable as in the above definition. In some real-world applications it may be necessary to adapt quality functions at run-time. This does not, however, affect the discussions on scheduling algorithms in the following chapter, as long as the functions are unambiguously defined at all times.

For a set of tasks, quality functions are grouped together in a vector for notational brevity. The set of such vectors is needed for the later definition of the objective function of quality / utility scheduling algorithms.

Definition 6 (Vectors of quality functions)
The set of vectors of all possible quality functions for the elements of a task set $\mathbb{T}' \subseteq \mathbb{T}$ is

$$\mathbb{QF}_{\mathbb{T}'} := \prod_{T \in \mathbb{T}'} \mathbb{QF}_T \qquad \textit{with} \qquad \mathbb{QF}_T := (\mathbb{R}_0^+)^{\mathbb{LT}_T}.$$ [1]

We use the notation $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$ for vectors of quality functions for all tasks in $\mathbb{T}'$.

The domain of $\vec{q}$ will always be unambiguous from the context without including the task set $\mathbb{T}'$ explicitly in the notation.

Note that there are no deadlines in our system, as urgency is expressed via utility functions. Therefore, tasks may conceptually be active for an infinitely long time. It is only for reasons of efficiency that a scheduling algorithm should remove tasks from consideration once they have low utility. Utility functions are defined on global time:

Definition 7 (Utility functions)
A utility function u_T for task $T \in \mathbb{T}$ is defined as a time-discrete function

$$u_T : \mathbb{GT} \rightarrow \mathbb{R}_0^+.$$

$u_T(t) = 0$ for all $t < r_T$ and $u_T(t)$ is monotonically decreasing for $t \geq r_T$.

[1]where for $\mathbb{T}' = \{T_1, \ldots, T_k\}$:

$$\prod_{T \in \mathbb{T}'} \mathbb{QF}_T := \mathbb{QF}_{T_1} \times \cdots \times \mathbb{QF}_{T_k}$$

Examples of utility functions can be seen in figure 2.4.

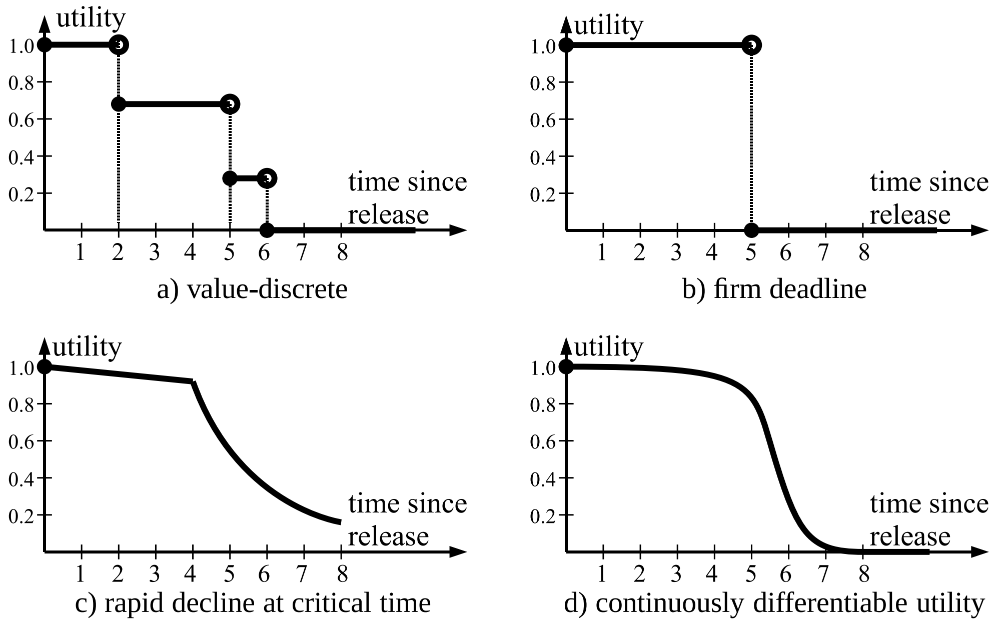


Figure 2.4: Example utility functions

Example a) represents a value-discrete utility function. Example b) models a firm deadline; if it is missed, the value gained from this task drops to zero, but no other effect to the environment is noticed. The utility function of c) shows a slow linear decline prior to and a rapid exponential decline after a critical time has been reached, and function d) is continuously differentiable.

Again, discrete utility functions defined on a discrete time domain are only approximations of the continuous example functions.

As with quality functions, utility functions for a set of tasks are grouped together in a vector for notational reasons, and we define the set of all possible such vectors.

Definition 8 (Vectors of utility functions)
The set of vectors of utility functions for $\mathbb{T}' \subseteq \mathbb{T}$ is

$$\mathbb{UF}_{\mathbb{T}'} := \prod_{T \in \mathbb{T}'} \mathbb{UF}_T \qquad \textit{with} \qquad \mathbb{UF}_T := (\mathbb{R}_0^+)^{\mathbb{GT}}.$$

We use the notation $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ for vectors of utility functions for all tasks in $\mathbb{T}'$.

Again, the domain of $\vec{u}$ will always be unambiguous from the context.

2.1.3 Local Time Functions

As shown above, a schedule is fully determined by giving the local times of all tasks for all global time instants. This leads us to the definition of local time functions.

Definition 9 (Local time functions)
For task $T \in \mathbb{T}$, a local time function is defined as monotonically increasing function

$$\tau_T : \mathbb{GT} \to \mathbb{LT}_T$$

with $\forall t \leq r_T : \tau_T(t) = 0$.
The set of all possible local time functions for T is denoted by $\mathbb{LTF}_T$.

The above condition means that no computation time can be allocated to a task prior to its release time.

Allocation functions are based on the same information as local time functions, but retrieve the amount of computation time allocated to tasks at each time instant. Obviously, within the model defined so far, allocation functions can only assume values of 0 or 1. In multiprocessor systems, further values are possible.

Definition 10 (Allocation functions)
The allocation function is defined as

$$\alpha_T : \mathbb{LTF}_T \times \mathbb{GT} \to \mathbb{LT}_T$$

with $\alpha_T(\tau_T, t) := \tau_T(t+1) - \tau_T(t)$.

For the set of tasks, the following must be true

$$\forall t \in \mathbb{GT} : 0 \leq \sum_{T \in \mathbb{T}} \alpha_T(\tau_T, t) \leq 1$$

$$\text{(especially: } \forall t \in \mathbb{GT} : \forall T \in \mathbb{T} : 0 \leq \alpha_T(\tau_T, t) \leq 1)$$

At any time instant no more than one task may be allocated the processor. Obviously, as time moves on, the progress of the task cannot decrease.

As a consequence, for the sum of local times:

$$\forall t \in \mathbb{GT} : 0 \leq \sum_{T \in \mathbb{T}} \tau_T(t) \leq t$$

$$\text{(especially: } \forall t \in \mathbb{GT} : \forall T \in \mathbb{T} : 0 \leq \tau_T(t) \leq t)$$

We define functions based on local time functions indicating how early a certain local time (level of progress) is reached by a task.

Definition 11 (Local timeliness functions)
For task $T \in \mathbb{T}$ with local time function $\tau_T \in \mathbb{LTF}_T$, the timeliness function $\underline{\tau}_T : \mathbb{LT}_T \to \mathbb{GT} \cup \{\infty\}$ is defined as:

$$\underline{\tau}_T(n) := \begin{cases} \min\{t \in \mathbb{GT} : \tau_T(t) \geq n\} & \textit{if there is such a } t \\ \infty & \textit{otherwise} \end{cases}$$

$\underline{\tau}_T(n)$ yields the earliest point of time when the associated local time function τ_T surpasses the value of n.

As before, we define vectors of local time functions and sets thereof for notational reasons.

Definition 12 (Vectors of local time functions)
For a subset $\mathbb{T}' \subseteq \mathbb{T}$ of tasks,

$$\mathbb{LTF}_{\mathbb{T}'} := \prod_{T \in \mathbb{T}'} \mathbb{LTF}_T$$

is the set of vectors of local time functions for tasks in $\mathbb{T}'$; we use the symbol $\vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$ for elements of this set.

The domain of $\vec{\tau}$ will always be unambiguous from the context.

2.1.4 Value Functions

For a finite task set $\mathbb{T}' \subseteq \mathbb{T}$, we know the quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$ and $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$. The question is how to select local time functions such that long-term reward is maximised. Any objective function defined for this purpose is based on the given quality and utility functions as well as the global time and the local time functions. As we are primarily interested in high long-term reward, we formulate that the goal of the scheduling algorithm is to find local time functions $\vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$ such that an objective function (which we will call *value function*)

$$v : \mathbb{QF}_{\mathbb{T}'} \times \mathbb{UF}_{\mathbb{T}'} \times \mathbb{LTF}_{\mathbb{T}'} \times \mathbb{GT} \to \mathbb{R}_0^+$$

for the parameters stated above and these local time functions is maximized for the global time $t \to \infty$.

The solution found by an optimal scheduler would be the vector of local time functions

$$\underset{\vec{\tau}\in\mathbb{LTF}_{\mathbb{T}'}}{\text{maxarg}} \lim_{t\to\infty} v_{\vec{q},\vec{u}}(\vec{\tau},t) \quad .$$

This statement includes that we assume the value function to be bounded and convergent for $t \to \infty$ with any parameter setting, such that the above limit always exists. For each task, identity is an upper bound to local time functions. Therefore, and because quality functions are monotonically increasing, $u_T(t) \cdot q_T(t)$ is an upper bound to the product of quality and utility $u_T(t) \cdot q_T(\tau_T(t))$ of task T for any local time function τ_T.

The way to combine quality and utility functions into a value function that we will use in our scheduling algorithms includes forming the pointwise product of these functions, as will be described in more detail in the following chapter.

Assume $\mathbb{T}' = \{T_1, T_2\}$ and figures 2.5a) and c) show the quality function and utility function of task T_1. Likewise, let figures 2.5b) and d) depict the quality and the utility functions of task T_2.

Figures 2.5e) and f) are the pointwise product of quality and utility functions of task T_1 and T_2, respectively, i.e., the current upper bounds $u_{T_1}(t) \cdot q_{T_1}(t)$ and $u_{T_2}(t) \cdot q_{T_2}(t)$ for the products of quality and utility $u_{T_1}(t) \cdot q_{T_1}(\tau_{T_1}(t))$ and $u_{T_2}(t) \cdot q_{T_2}(\tau_{T_2}(t))$.

Defining the current time to be 0, for local time functions τ_{T_1} and τ_{T_2}, the sum of the products of quality and utility for the two tasks at time t is

$$u_{T_1}(t) \cdot q_{T_1}(\tau_{T_1}(t)) + u_{T_2}(t) \cdot q_{T_2}(\tau_{T_2}(t)).$$

Of course, resource constraints apply to the task set, i.e.

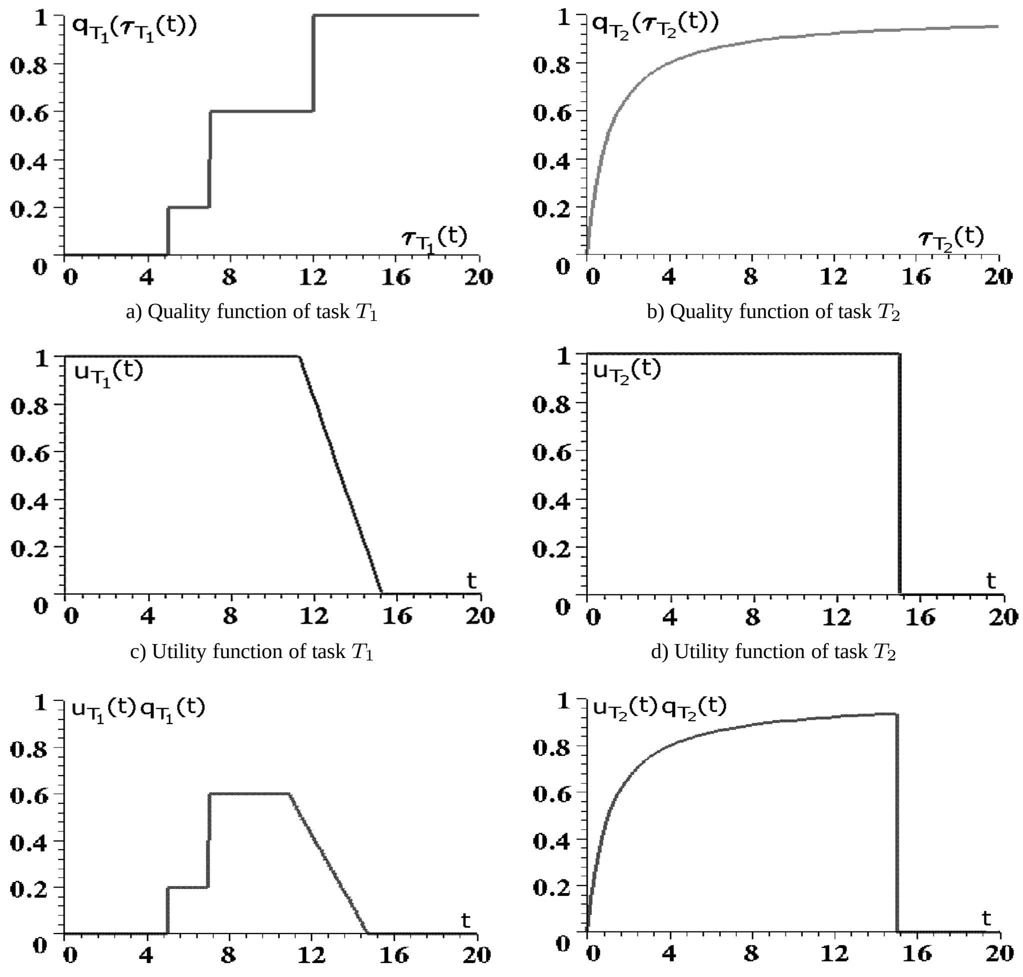
$$\alpha_{T_1}(\tau_{T_1}, t) + \alpha_{T_2}(\tau_{T_2}, t) \leq 1.$$

As $\tau_{T_1}(t) \leq t$, $\tau_{T_2}(t) \leq t$, quality functions are monotonically increasing and utility functions are non-negative, we now see that

$$u_{T_1}(t) \cdot q_{T_1}(t) + u_{T_2}(t) \cdot q_{T_2}(t)$$

is an upper bound for the above expression. However, this is not very useful, as it does not give us any hint for an appropriate distribution of resources. Therefore, instead of looking at the upper bound of the sum of products of quality and utility, we now investigate the sum of the upper bounds of products of quality and utility, receiving a function of two time domains:

$$u_{T_1}(t_1) \cdot q_{T_1}(t_1) + u_{T_2}(t_2) \cdot q_{T_2}(t_2)$$



a) Quality function of task T_1

b) Quality function of task T_2

c) Utility function of task T_1

d) Utility function of task T_2

e) Product of quality and utility functions for task T_1

f) Product of quality and utility functions for task T_2

Figure 2.5: Forming pointwise products of quality and utility functions

Figure 2.6 shows the resulting profile for the example functions.

Assuming once again the processor can be fully utilised by the task set, we may in fact use this profile to find actual distributions of processor time among the tasks. We note that the resource constraint takes the following form for $t \in \mathbb{GT}$ when applied to the upper bounds of the resource allocations:

$$t_1 + t_2 \leq t$$

Some diagrams for various values of t are shown in figure 2.7. A search algorithm is subsequently applied to find maxima in these problem spaces.

Based on the prior assumptions of upper bounds for the product of quality and utility as well as the full utilisation of processing time by the task set, it can now be justified to identify local

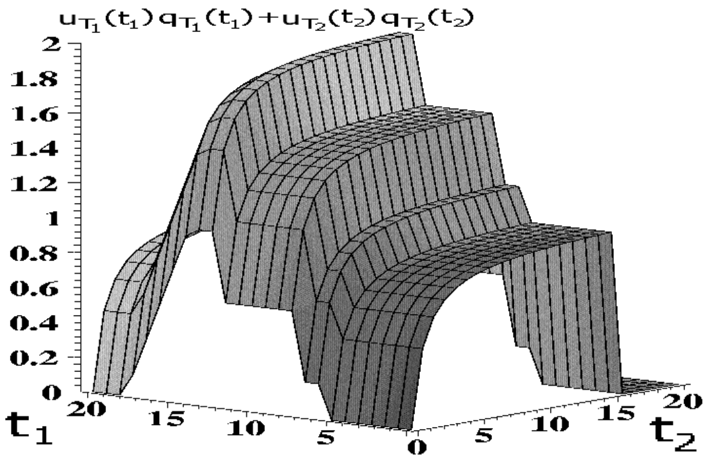
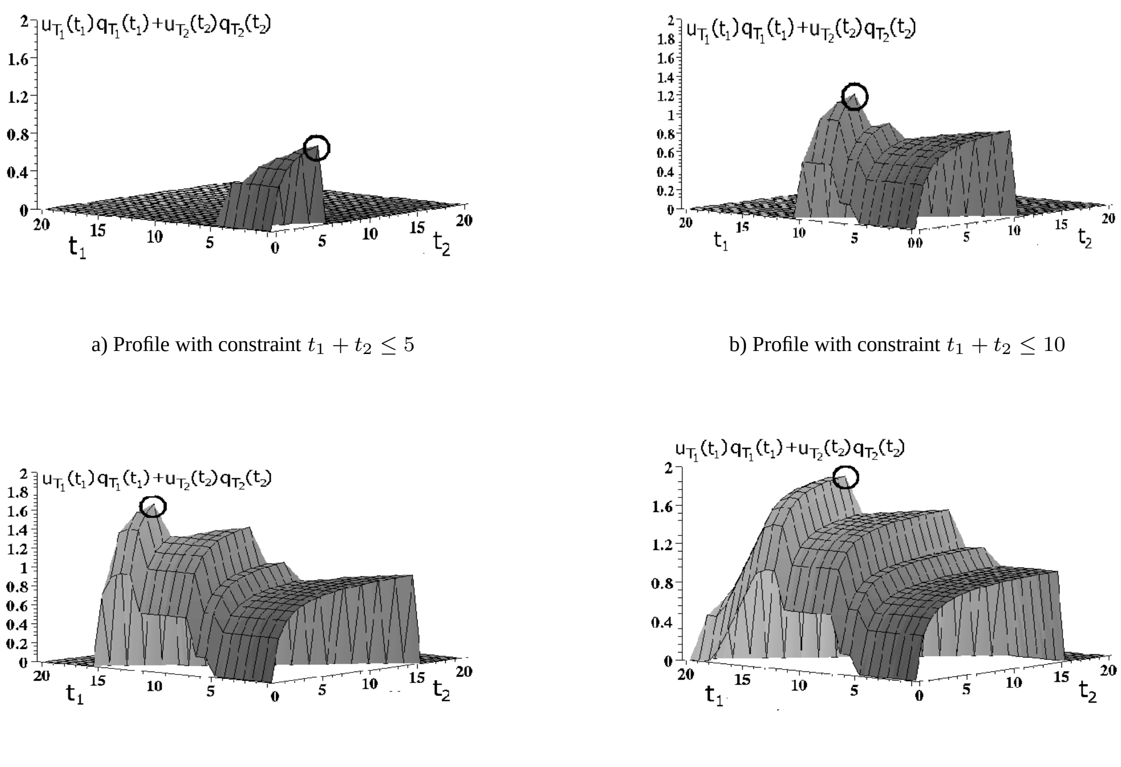


Figure 2.6: Profile $u_{T_1}(t_1) \cdot q_{T_1}(t_1) + u_{T_2}(t_2) \cdot q_{T_2}(t_2)$



a) Profile with constraint $t_1 + t_2 \leq 5$

b) Profile with constraint $t_1 + t_2 \leq 10$

c) Profile with constraint $t_1 + t_2 \leq 15$

d) Profile with constraint $t_1 + t_2 \leq 20$

Figure 2.7: Profiles with constraints $t_1 + t_2 \leq t$ for $t \in \{5, 10, 15, 20\}$

times with the values t_1 and t_2, so that by connecting the maximum values gained for search spaces with various values of t we receive the value function graph for $\tau_{T_1}(t) + \tau_{T_2}(t)$ in figure 2.8. Fortunately, the scheduler only needs to evaluate these functions at a small number of points, so the value function does not have to be calculated for a large interval of time, as the figure might suggest.

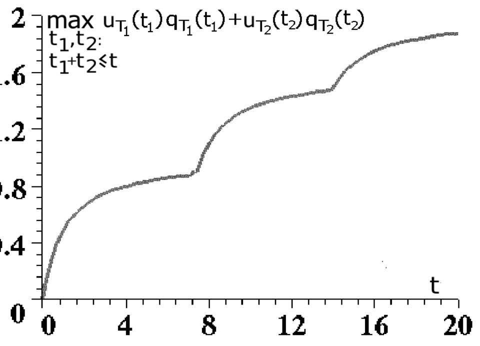


Figure 2.8: Value function

2.2 Dynamic Scheduling

The dynamic quality / utility scheduling scheme we propose means that partial schedules are generated in a series of consecutive phases at not necessarily equidistant times. In these scheduling phases, release times are estimated for tasks likely to arrive within a limited-size time window into the future. We assume that within any finite interval of time, only a finite number of new tasks may arrive. Quality and utility functions of tasks already released earlier are updated: actual release times are now used instead of the estimates, and the quality functions are transformed according to the processor time that has already been allocated to the task in the past. The partial schedules that are calculated for these time windows may have to be adapted or recalculated even before the end of the window if it turns out that the release time estimates were too far from reality.

Figure 2.9 shows the core components of the scheduling architecture we use. Data known before runtime are the quality and utility functions of all tasks as well as stochastic distributions for the release times of future tasks. These data are passed on to the dynamic scheduler along with the definition of a suitable value function. At runtime, the dynamic scheduler estimates the release times of future tasks, decides on a set of tasks to investigate and uses a search technique to find an appropriate allocation of processor time to the tasks so that the objective function is optimised. The resulting partial schedules are translated into individual task allocation and revocation actions. Data required to be provided by the environment (which in our case is a simulation system) is information on the actual behaviour of tasks (primarily their release times that have previously been estimated) and a time signal to drive the scheduling and dispatching process. Based on the actual data on the release times of tasks, the scheduler has to decide if and when to adapt or recalculate schedules.

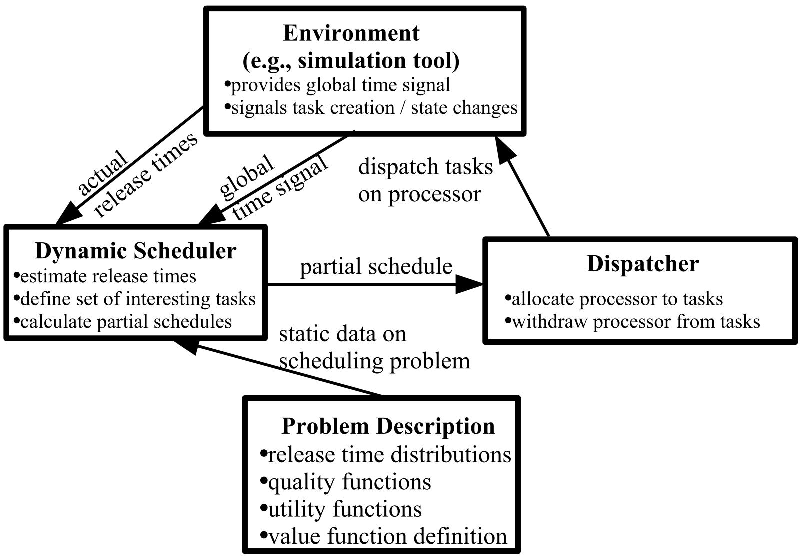

Environment
(e.g., simulation tool)
•provides global time signal
•signals task creation / state changes
actual release times
global time signal
dispatch tasks on processor
Dynamic Scheduler
•estimate release times
•define set of interesting tasks
•calculate partial schedules
partial schedule
Dispatcher
•allocate processor to tasks
•withdraw processor from tasks
static data on scheduling problem
Problem Description
•release time distributions
•quality functions
•utility functions
•value function definition

Figure 2.9: Core components of scheduling architecture

2.3 Time-Variant Value Functions

Value functions are functions of several time domains and hence are not time-invariant with respect to the global timeline. The following example demonstrates the shape of the upper bound of a value function during the “life time” of a task T, starting from its release time r_T. In the following example, we use the value function

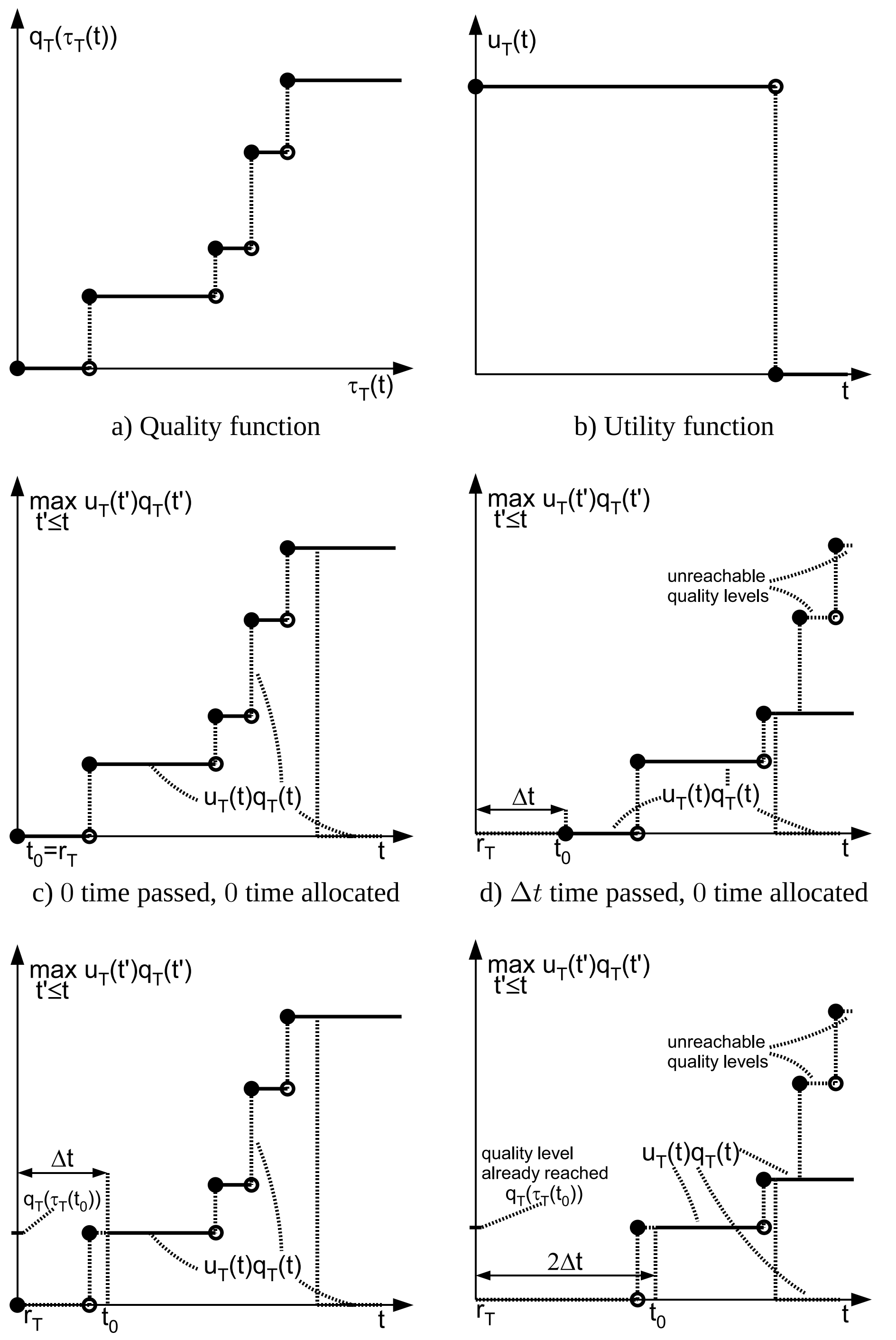
$$\max_{t' \leq t} u_T(t') \cdot q_T(\tau_T(t'))$$

(which is one of the variants we will introduce later in this chapter).

Let the quality function be piecewise constant (figure 2.10a)) and the utility function represent a firm deadline (figure 2.10b)).

Figure 2.10c) shows the upper bound

$$\max_{t' \leq t} u_T(t') \cdot q_T(t')$$



a) Quality function b) Utility function

c) 0 time passed, 0 time allocated d) Δt time passed, 0 time allocated

e) Δt time passed, Δt time allocated f) $2 \cdot \Delta t$ time passed, Δt time allocated

Figure 2.10: Time variance of example value function

of the value function with the current time t_0 equaling the release time of the task. Figure 2.10d) depicts the situation when Δt (global) time has passed, but no cpu time at all has been allocated to the task. The two highest quality levels are already unreachable now, because even if the task receives exclusive service from now on, it cannot possibly calculate long enough before its deadline (i.e., its decline in utility). On the contrary, in figure 2.10e) the task was able to execute during the whole interval $[r_T; r_T + \Delta t[$. The task has already reached the first positive quality level $(q_T(\tau_T(t_0)))$ and still has the potential to reach its highest levels. Finally, in figure 2.10f) $2 \cdot \Delta t$ time has passed, and Δt units of cpu time have been allocated to the task. Both of the effects described above can be noticed here. The figures demonstrate that in the (normal) case when a task does not receive full allocation of the processor within any interval of time, quality levels with high resource requirements become more and more unreachable.

Let us now investigate the shape of the search profile from which a value function is calculated for a set of two tasks $\{T_1, T_2\}$ with the quality and utility functions of figure 2.5 at a later point in time, namely after 10 time units. Like in the previous example for a single task, the search profile does not remain constant over time. Assume the tasks are released at the same time and the processor time can be fully distributed among the two tasks during this interval. The development of the search profile depends on this distribution of resources. First, let all 10 units of time have been allocated to task T_1. In this case only the quality of task T_1 was able to advance, as figures 2.11a) and 2.11b) show. The situation of figure 2.10e) applies to task T_1, and the situation of figure 2.10d) to task T_2. Figures 2.11c) and 2.11d) show the utility functions with the vertical line indicating the (global) time passed since the release of the tasks. Intuitively, current time has approached the (firm and soft) deadlines of both tasks by 10 units of time.

The resulting profile from which to calculate the value function is given in figure 2.12b).

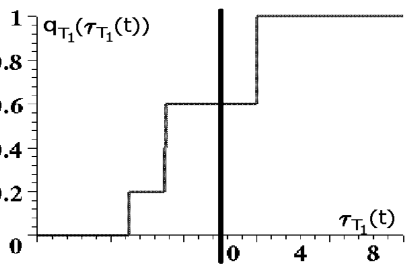
On the other hand, now assume all 10 units of time were allocated to task T_2. Figure 2.12c) shows the profile for this case. Now the situation of figure 2.10d) applies to task T_1 and the situation of figure 2.10e) to task T_2.

Finally, in figure 2.12d) each of the tasks was allowed to execute for 5 of the 10 time units, i.e., both tasks encounter the situation of figure 2.10f).

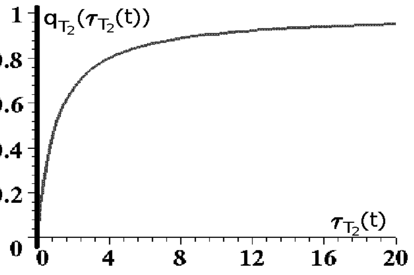
2.4 Properties of Value Functions

This section describes a set of properties we assume to be valid for all value functions used within the scheduling framework outlined in the previous chapter. Later, we propose a series of example value functions with these properties.

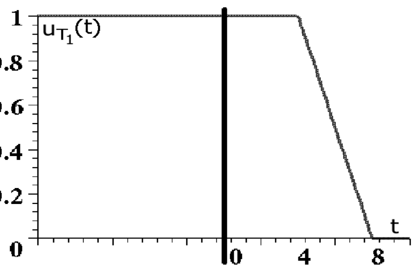
For a finite set of tasks $\mathbb{T}' \subseteq \mathbb{T}$, a vector $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$ of quality functions, and a vector $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$



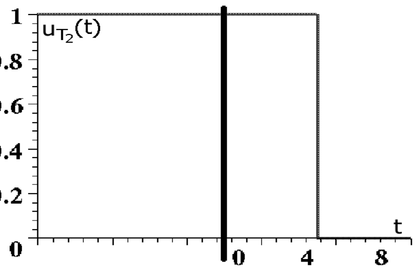
a) Quality function of task T_1



b) Quality function of task T_2

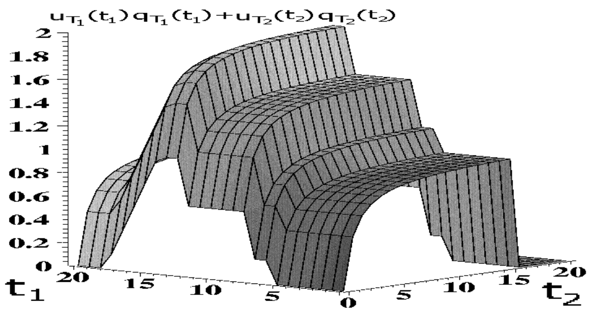


c) Utility function of task T_1

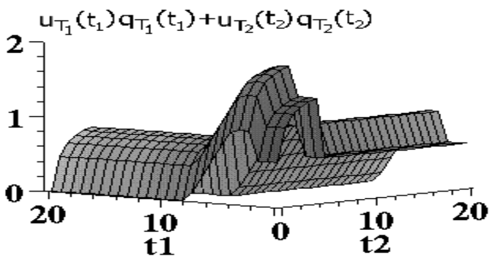


d) Utility function of task T_2

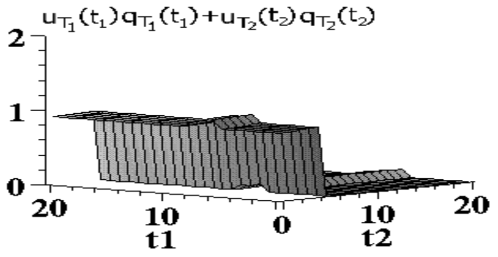
Figure 2.11: Quality and utility functions after 10 time units with full allocation to task T_1



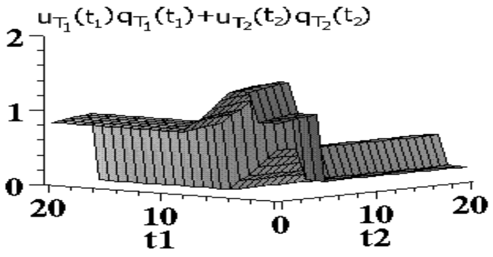
a) at the common release time of T_1 and T_2



b) after 10 units with full allocation to T_1



c) after 10 units with full allocation to T_2



d) after 10 units with fair share between T_1 and T_2

Figure 2.12: Search profiles

of utility functions, we want to find local time functions $\vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$, such that a value function

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t)$$

is maximized for $t \to \infty$. Remember that the vector of local time functions is a complete description of the schedule, providing information on processor allocation for every point in time and every task in the task subset of interest $\mathbb{T}'$. We implicitly assume that no processor time is allocated to tasks outside this set. The preparatory step we are not going to deal with explicitly is the selection of an appropriate task set; suffice to say we include tasks that have either already been released or will be released in the near future. Additionally, the scheduler may exclude tasks with low utility.

In order to find appropriate value functions, we are first going to define characteristics by stating a series of conditions we require objective functions to hold. We assume that for all objective functions, the following is true:

(1) Global time monotony: A longer execution time of the system results in a higher or equal overall value; therefore, any value function must be monotonically increasing in t, i.e., for a given task set $\mathbb{T}'$ with quality functions $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, utility functions $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$, local time functions $\vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$, and time $t \in \mathbb{GT}$:

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) \leq v_{\vec{q},\vec{u}}(\vec{\tau}, t+1)$$

(2) Allocation history monotony: If two vectors of local time functions $\vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$ represent the same resource allocation to the tasks up to a certain time t_1, the value up to this time is the same for both vectors of local time functions. In other words, a value function must be prefix monotonic in the vector of local time functions, i.e., for a given task set $\mathbb{T}'$ with quality functions $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, utility functions $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ and time $t_1 \in \mathbb{GT}$:

$$\forall t \in \{0, \ldots, t_1\} : \forall T \in \mathbb{T}' : \tau_T(t) = \tau'_T(t)$$
$$\Rightarrow \forall t \in \{0, \ldots, t_1\} : v_{\vec{q},\vec{u}}(\vec{\tau}, t) = v_{\vec{q},\vec{u}}(\vec{\tau'}, t)$$

(3) Allocation amount monotony: A higher allocation of resources to a task means a higher or equal value of the overall system, i.e., a value function must be monotonically increasing in every local time function τ_T. Let $\vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$ be two vectors of local time functions and assume there is a $T' \in \mathbb{T}'$ such that

- $\forall t \in \mathbb{GT} : \tau_{T'}(t) \leq \tau'_{T'}(t)$

- $\forall T \in \mathbb{T}' \backslash \{T'\} : \forall t \in \mathbb{GT} : \tau_T(t) = \tau'_T(t)$

Then for the value function the following must be true for $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ and $t \in \mathbb{GT}$:

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) \leq v_{\vec{q},\vec{u}}(\vec{\tau'}, t)$$

(4) Allocation time monotony: An earlier allocation of resources to tasks means a higher or equal value of the overall system, i.e., a value function must be monotonically decreasing in every local timeliness function $\underline{\tau}_T$:

Let $\vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$ be vectors of local time functions and assume there is a $T' \in \mathbb{T}'$ such that

- $\forall n \in \mathbb{LT}_{T'} : \underline{\tau}_{T'}(n) \geq \underline{\tau}'_{T'}(n)$
- $\forall T \in \mathbb{T}' \backslash \{T'\} : \forall n \in \mathbb{LT}_T : \underline{\tau}_T(n) = \underline{\tau'}_T(n)$

Then for quality functions $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, utility functions $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ and time $t \in \mathbb{GT}$:

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) \leq v_{\vec{q},\vec{u}}(\vec{\tau'}, t)$$

(5) Reducibility to utility intervals: Dividing global time into intervals without utility change for any task, for arbitrary quality functions and local time functions, the value is fully determined by local times at the ends of these intervals.

Let $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}, \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$ be vectors of local time functions and assume there are $t_1, t_2 \in \mathbb{GT}$ with $t_1 < t_2$ such that for all tasks $T \in \mathbb{T}'$:

$$u_T(t_1) = u_T(t_1 + 1) = \cdots = u_T(t_2)$$

$$\forall t \leq t_1 : \tau'_T(t) = \tau_T(t)$$

and

$$\tau'_T(t_2) = \tau_T(t_2)$$

Then for the value function at time t_2 with $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$:

$$v_{\vec{q},\vec{u}}(\vec{\tau'}, t_2) = v_{\vec{q},\vec{u}}(\vec{\tau}, t_2)$$

(6) Utility monotony: Higher utility of tasks results in higher or equal value of the system, i.e., a value function must be monotonically increasing with any utility function of tasks in the following sense:

Let $\mathbb{T}', \mathbb{T}'' \subseteq \mathbb{T}$ and assume there are $T' \in \mathbb{T}'$ and $T'' \in \mathbb{T}''$ such that

$$r_{T'} = r_{T''} \quad \text{and} \quad q_{T'} = q_{T''} \quad \text{and} \quad \forall t \in \mathbb{GT} : u_{T'}(t) \leq u_{T''}(t)$$

and $\mathbb{T}' \backslash \{T'\} = \mathbb{T}'' \backslash \{T''\}$. [2]

Then for all local time functions $\vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}, \vec{\tau''} \in \mathbb{LTF}_{\mathbb{T}''}$ with $\tau'_{T'} = \tau''_{T''}, \vec{q'} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u'} \in \mathbb{UF}_{\mathbb{T}'}, \vec{q''} \in \mathbb{QF}_{\mathbb{T}''}, \vec{u''} \in \mathbb{UF}_{\mathbb{T}''}$, and $t \in \mathbb{GT}$, the following must be true:

$$v_{\vec{q'},\vec{u'}}(\vec{\tau'}, t) \leq v_{\vec{q''},\vec{u''}}(\vec{\tau''}, t)$$

(7) Quality monotony: Higher quality of tasks results in higher or equal value of the system, i.e., a value function must be monotonically increasing with any quality function of tasks in the following sense:

Let $\mathbb{T}', \mathbb{T}'' \subseteq \mathbb{T}$ and assume there are $T' \in \mathbb{T}'$ and $T'' \in \mathbb{T}''$ such that

$$r_{T'} = r_{T''}, u_{T'} = u_{T''} \quad \text{and} \quad \forall n' \in \mathbb{LT}_{T'}, n'' \in \mathbb{LT}_{T''} : n' \equiv n'' \Rightarrow q_{T'}(n') \leq q_{T''}(n'')$$

and $\mathbb{T}' \backslash \{T'\} = \mathbb{T}'' \backslash \{T''\}$.

Then for all local time functions $\vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}, \vec{\tau''} \in \mathbb{LTF}_{\mathbb{T}''}$ with $\tau'_{T'} = \tau''_{T''}, \vec{q'} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u'} \in \mathbb{UF}_{\mathbb{T}'}, \vec{q''} \in \mathbb{QF}_{\mathbb{T}''}, \vec{u''} \in \mathbb{UF}_{\mathbb{T}''}$, and $t \in \mathbb{GT}$, the following must be true:

$$v_{\vec{q'},\vec{u'}}(\vec{\tau'}, t) \leq v_{\vec{q''},\vec{u''}}(\vec{\tau''}, t)$$

Note that properties 3 (allocation amount monotony) and 4 (allocation time monotony) are actually equivalent; nevertheless, they are stated separately, because in different problem settings, one of them may be easier to prove than the other. To see that the two properties are equivalent, note that from the definition of timeliness functions, we know that

$$\underline{\tau}_T(\tau_T(t)) = \min\{t' \in \mathbb{GT} : \tau_T(t') \geq \tau_T(t)\} \leq t$$

and

$$\tau_T(\underline{\tau}_T(n)) = \tau_T(\min\{t' \in \mathbb{GT} : \tau_T(t') \geq n\}) = n$$

[2]Of course, local time domains of T' and T'' are isomorphic, so that $q_{T'}$ and $q_{T''}$ can be compared.

Then $\underline{\tau}'_T(t) \leq \underline{\tau}_T(t)$ is equivalent to

$$\tau_T(t) = (id_{\mathbb{L}\mathbb{T}_T} \circ \tau_T)(t) = ((\tau'_T \circ \underline{\tau}'_T) \circ \tau_T)(t) = \tau'_T(\underline{\tau}'_T(\tau_T(t))) \leq \tau'_T(\underline{\tau}_T(\tau_T(t))) \leq \tau'_T(t)$$

Therefore, for any given value function, we only need to show that it conforms with one of the two properties.

2.5 Example Value Functions

In this section we will introduce a few example value functions that hold the conditions of the previous section.

2.5.1 Pointwise Sum of Product of Utility and Quality Functions with Outer Hold Operator

As a first example, we want to define the objective function as the sum of the pointwise product of quality and utility functions of individual tasks

$$\sum_{T \in \mathbb{T}'} u_T(t) \cdot q_T(\tau_T(t)),$$

as was already mentioned in the previous chapter, where the quality function is evaluated according to the local time of this task, and the utility function is evaluated for the global time.

To take into account the property of the objective function being monotonically increasing with the global time, we add a “hold operator” (the maximum value of the original function encountered up to some given time), such that the value function renders as follows:

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t'))$$

The proof that this definition complies with the properties of value functions stated above can be found in appendix B.1.

2.5.2 Pointwise Sum of Product of Utility and Quality Functions with Inner Hold Operator

Instead of applying a hold operator to the entire sum

$$\sum_{T \in \mathbb{T}'} u_T(t) \cdot q_T(\tau_T(t)),$$

we can apply it to individual tasks and sum up afterwards:

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \sum_{T \in \mathbb{T}'} \max_{t' \leq t} u_T(t') \cdot q_T(\tau_T(t'))$$

The proof that this definition complies with the properties of value functions stated above can be found in appendix B.2.

2.5.3 Pointwise Sum of Product of Quality and Utility Functions with Additional Conditions

An alternative to the hold operator is to pose additional preconditions on the quality and utility functions as well as on the local time functions. Consider the following function:

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) = \sum_{T \in \mathbb{T}'} u_T(t) \cdot q_T(\tau_T(t))$$

with the constraint

$$\forall T \in \mathbb{T} : \forall t \in \mathbb{GT} : u_T(t+1) \cdot q_T(\tau_T(t+1)) \geq u_T(t) \cdot q_T(\tau_T(t))$$

The proof that this definition complies with the properties of value functions stated above can be found in appendix B.3.

Let us now look at some examples for task sets satisfying the additional constraint.

2.5.3.1 Background Anytime Tasks

Background tasks do not have any timing constraints; in our model, they have constant utility functions. Let $u_T(t) = 1$ for all $T \in \mathbb{T}'$ and $t \in \mathbb{GT}$ and $q_T : \mathbb{LT}_T \to \mathbb{R}_0^+$ be monotonically increasing. Then

$$\begin{aligned} u_T(t+1) \cdot q_T(\tau_T(t+1)) &= q_T(\tau_T(t+1)) \\ &\geq q_T(\tau_T(t)) \\ &= u_T(t) \cdot q_T(\tau_T(t)) \end{aligned}$$

2.5.3.2 Tasks with Mandatory and Optional Service Times

Another suitable class of tasks for this category of value functions requires to be serviced at specific times, whereas service at other times is optional; note that this property of mandatory and optional service times is different from mandatory and optional parts of computations. Assume task $T_i \in \{T_1, \ldots, T_k\}$ with

- release times $r_{T_1} = \cdots = r_{T_k} = 0$ for simplicity
- utility functions for $t \in \mathbb{GT}$

$$u_{T_i}(t) = \begin{cases} 1 & \text{if } t < 2^i \\ \frac{1}{2} & \text{if } t \geq 2^i \end{cases}$$

- quality functions for $n \in \mathbb{LT}_{T_i}$:

$$q_{T_i}(n) = \begin{cases} 0 & \text{if } n = 0 \\ (\frac{1}{3})^{2^i - n} & \text{if } 1 \leq n \leq 2^i \\ 1 & \text{if } n > 2^i \end{cases}$$

- the service guarantee

$$\forall i \in \mathbb{N}_0, n \geq 1 : \alpha_{T_i}(\tau_{T_i}, 2^i - 1) = 1$$

The system has to guarantee to schedule each task for execution at least at the instant in time when its utility drops; note that this guarantee is possible in this specific setting, as no two tasks change their utilities at the same time. The proof of these definitions complying with the additional properties given above can be found in appendix B.4.

2.5.4 Pointwise Maximum of Product of Utility and Quality Functions

Define the value function as follows

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t'))$$

The proof that this definition complies with the properties of value functions stated above can be found in appendix B.5.

Chapter 3

Scheduling Algorithms

There cannot be a crisis next week. My schedule is already full.
Henry Kissinger

In this chapter we suggest several algorithms for the basic quality / utility scheduling problem. As mentioned before, we need to develop dynamic scheduling algorithms to run concurrently with the application tasks to be scheduled. The dynamic scheduler is invoked repeatedly at run time and works on both a set of tasks that have already arrived and set of tasks which are likely to be released in the near future. Obviously, the ability of a scheduling algorithm to predict the release times of tasks correctly determines its performance to a large extent. Of course, this ability depends on the probability distributions of the task release times which are supposed to be known in advance. The higher the *variance* in the release time distributions, the more difficult it is to accurately estimate the set of future ready tasks. Furthermore, the larger the *time horizon* for the prediction, the better the resource distribution to future tasks can potentially be. In the extreme case, taking a myopic approach, no prediction of the future behaviour is made at all, and all scheduling decisions are based entirely on tasks having been released in the past. Scheduling takes place in a series of consecutive *phases*, and the resulting partial schedules remain valid until the beginning of the next scheduling phase. For each scheduling phase, we assume that the application may utilise the entire set of resources (up to now: a single processor) to full extent. Having introduced the terms of phase and horizon, we can now clarify the aim of the scheduler to distribute the processing time among the set of tasks made up of the ones released prior to the beginning of the scheduling phase and those likely to be released between the beginning of the scheduling phase and the end of the time horizon. We call the interval of time between the beginning of a phase and the end of the corresponding time horizon the *scheduling window* associated with this phase. We generally assume that the length of the windows remains approximately constant for all phases, although changes can be made depending on the behaviour of the system. Obviously, the window size largely influences the performance of the scheduler. A small window gives the scheduler too little insight into the estimated future behaviour of the application, so that it can hardly make any sensible decisions. On the other hand, larger window sizes increase the overhead caused by the online optimisation; as a dynamic scheduler competes with the application tasks for the shared resources, a degradation of system performance can be expected.

It is not obvious, however, which optimisation or search algorithm is best suited for the computation of good distributions of processor time among the tasks. An important piece of information in this regard is the reliability of the estimation of future task release times. If the set of ready tasks within each interval of time can be predicted with high certainty or even deterministically, the main focus of the scheduling algorithm can be directed towards finding locally optimal distributions without paying too much attention to the cost of contingency actions like rescheduling or schedule adaptation. Contingency actions can be allowed to be expensive if the probability of having to trigger them is sufficiently low. On the other hand, if the scheduler has to deal with higher levels of uncertainty, it may be more favourable to divert some of the effort awarded to scheduling to the provision of alternative plans for the set of most likely situations than can occur at a certain time. It appears natural that calculating schedules for sets of possible situations is more time-consuming than for a deterministically known single future state. Hence, for a comparable level of scheduling effort, the former scheme will in general only be able to find inferior schedules for each situation investigated than the latter one, which can concentrate on one specific situation.

We can expect a tradeoff between the cost of contingency actions in a scheduling scheme and the quality of schedules for individual situations, driven by the probability that such actions become necessary. In this context, we may also define ranges of acceptability of schedules with regard to the deviation of the current situation from a prior estimate. This is to mean that rescheduling or schedule adaptation may not necessarily be triggered by each minor change in release times, based on the assumption that solutions to similar problems are similar to solutions of the original problem. This important assumption of all local-search techniques is of course not always valid due to possible discontinuities in the search space.

We first derive a dynamic scheduling scheme which iteratively estimates the single most likely situation in terms of ready tasks with associated release times for a fixed-size time window into the future. The original problem description is formulated in a way that makes it applicable to local-search algorithms like Simulated Annealing and meta-heuristic algorithms like Tabu

Search. We use thresholds to limit the set of ready tasks to those with sufficiently high utility and to identify major deviations of release times from their prior estimate, making it necessary to trigger schedule adaptation or rescheduling. In general, however, the search algorithms used are not able to build on prior solutions and thus adapt schedules, so that we have to resort to recalculating the current partial schedule in such cases. Unfortunately, rescheduling is expensive, so that the scheme can only be expected to perform well if the release times can be estimated with high accuracy. We call this scheme *reactive*, because it can only react to unlikely future situations when they occur without taking any prior precautions for them. Local-search algorithms have previously been applied in the context of scheduling problems, and Tabu Search schemes can be found for scheduling and related time-based planning problems. However, we are not aware of their prior use within dynamic flexible schedulers. The decisive step was to restrict the problem space to gain a finite description and thus enable these search algorithms to be applied in a dynamic setting to receive partial schedules, even though we must be aware that limited search spaces cannot be expected to result in globally optimal schedules even for deterministic problem settings with entirely known characteristics.

The second scheme we devised codes sets of most likely encountered future states with associated partial schedules into Markov Decision Processes (MDPs). The set of ready tasks is not limited by a fixed-size window, but by the likeliness of their release. The dynamic scheduler simulates the release behaviour of the task set and constructs a weighted state transition graph originating from the current situation with transition probabilities as weights. Again, thresholds are applied to exclude low-utility tasks. Now, contingency actions are far less likely to be needed, but computing schedules for alternative situations is also time-consuming. This additional effort pays off if release times cannot be predicted with high accuracy. We call this scheme *proactive*, because it provides strategies for dealing with situations before we know they occur. The application of decision-theoretic reasoning to dynamic value-based real-time scheduling appears to be novel. Our approach is based on analogies between the search spaces in stochastic route planning and dynamic real-time scheduling.

3.1 Reactive Unconditional Scheduling

The first set of scheduling algorithms works directly on a limited-size window of time into the future for each scheduling phase as described above to determine a set of tasks which are likely to be released within this period of time. Figure 3.1 shows the development of the first three scheduling phases for an example application. Invocations of the dynamic scheduler are interleaved with execution of application tasks. One of the most important levers for optimising the system performance is the selection of a reasonable amount of processor time awarded to the scheduling algorithm. In many respects, the dynamic scheduler can be seen as just an additional task to the ones given in the problem description, although it does in reality require some special handling. Embedding the scheduler task into the task set almost naturally leads to the alternation of processor allocation between scheduler and application tasks (very much like the context switch between two application tasks) and the schedule being gradually developed by appending successive partial schedules. The filled circles represent the (real or estimated) release times of tasks, the contiguous lines the period of activity (i.e., tasks are ready for execution) of tasks with known release times, and the dotted lines the estimated period of activity of tasks with hitherto unknown release times.

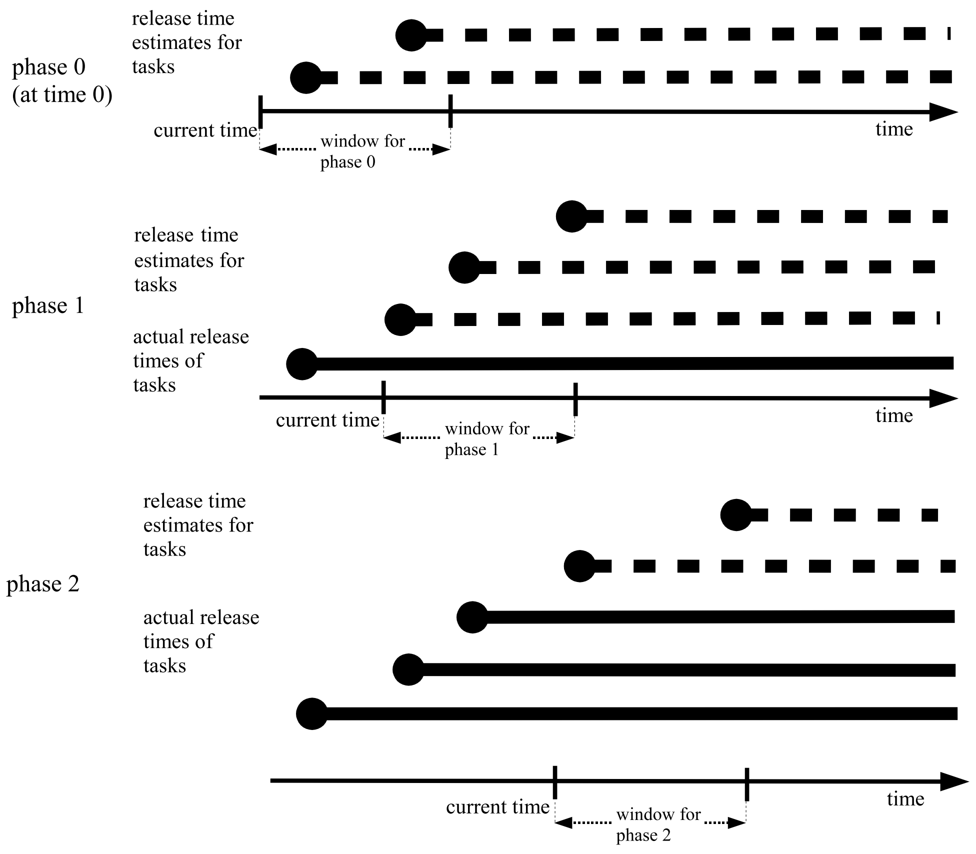

phase 0
(at time 0)
release time estimates for tasks
current time
window for phase 0
time
phase 1
release time estimates for tasks
actual release times of tasks
current time
window for phase 1
time
phase 2
release time estimates for tasks
actual release times of tasks
current time
window for phase 2
time

Figure 3.1: Scheduling phases

3.1.1 Determining the Task Set

Whenever the dynamic scheduler is invoked, the current set of tasks is enlarged such as to include the tasks which are likely to be released until the end of the scheduling window. Note that scheduling windows may (and in fact usually do) overlap. The system may opt for adaptation of the current partial schedule or even complete rescheduling if the estimated release times differ largely from their actual values. Furthermore, because the optimisation of processor allocation is cut off at this boundary, the scheduler does not treat favourably tasks that arrive shortly before the end of the scheduling window. The largest amount of potential service to these tasks will probably be beyond the end of the window and for this reason resource allocation to these tasks does not appear to the scheduler to be very rewarding. For this reason, it is generally necessary to start the next scheduling phase well ahead of the end of the window in figure 3.1. Note that the lines representing the individual tasks have a defined start (the release time), but no end; as mentioned earlier, tasks may conceptually be executed for an arbitrarily long time, and it is the responsibility of the scheduling algorithm to dismiss tasks which are unlikely to contribute much to the system performance (presumably those with low utility).

As an example for the remainder of this section, assume that the scheduler estimates that three tasks T_1, T_2 and T_3 will be released within the scheduling window, and their parameters are:

$$r_{T_1} = 0 \qquad u_{T_1}(t) = \begin{cases} 0 & \text{if } t - r_{T_1} < 0 \\ 1 & \text{if } 0 \le t - r_{T_1} < 6 \\ 0.6 & \text{if } 6 \le t - r_{T_1} < 13 \\ 0.1 & \text{if } t - r_{T_1} \ge 13 \end{cases} \qquad q_{T_1}(n_1) = \begin{cases} 0 & \text{if } 0 \le n_1 < 4 \\ 0.3 & \text{if } 4 \le n_1 < 8 \\ 0.4 & \text{if } 8 \le n_1 < 12 \\ 0.8 & \text{if } n_1 \ge 12 \end{cases}$$

$$r_{T_2} = 2 \qquad u_{T_2}(t) = \begin{cases} 0 & \text{if } t - r_{T_2} < 0 \\ 1 & \text{if } 0 \le t - r_{T_2} < 8 \\ 0.2 & \text{if } 8 \le t - r_{T_2} < 12 \\ 0 & \text{if } t - r_{T_2} \ge 12 \end{cases} \qquad q_{T_2}(n_2) = \begin{cases} 0 & \text{if } 0 \le n_2 < 2 \\ 0.1 & \text{if } 2 \le n_2 < 4 \\ 0.2 & \text{if } 4 \le n_2 < 6 \\ 0.3 & \text{if } n_2 \ge 6 \end{cases}$$

$$r_{T_3} = 5 \qquad u_{T_3}(t) = \begin{cases} 0 & \text{if } t - r_{T_3} < 0 \\ 1 & \text{if } 0 \le t - r_{T_3} < 8 \\ 0.7 & \text{if } 8 \le t - r_{T_3} < 10 \\ 0.1 & \text{if } t - r_{T_3} \ge 10 \end{cases} \qquad q_{T_3}(n_3) = \begin{cases} 0 & \text{if } 0 \le n_3 < 2 \\ 0.4 & \text{if } 2 \le n_3 < 8 \\ 1.0 & \text{if } n_3 \ge 8 \end{cases}$$

$$t \in \mathbb{GT}, \qquad n_1 \in \mathbb{LT}_{T_1}, \qquad n_2 \in \mathbb{LT}_{T_2}, \qquad n_3 \in \mathbb{LT}_{T_3}$$

3.1.2 Local-Search Approach

One possibility for solving the optimisation problem of maximising the value function is by heuristic local-search methods. Local-search methods are based on the neighbourhood assumption, i.e., on the perception that similar problem settings have similar solutions. In our case, search steps are described in terms of units of processor time allocated to individual tasks. We developed this scheme with the objective of possibly reusing previously calculated solutions at a later time. A major advantage is that the scheme can be adapted for a wide range of local-search algorithms without additional reasoning about the specific requirements of the scheduling problem. A general framework abstracts from these problem-specific properties and leaves the optimisation algorithms with a rather generic search problem, the quality and utility attributes as well as the release times and resource constraints coded into the search space and the objective function.

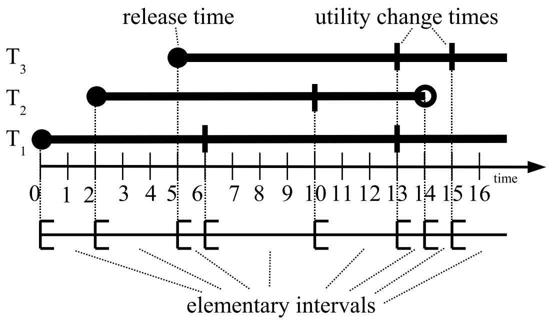
3.1.2.1 Calculation of Elementary Intervals

In a preparatory step the scheduler represents the problem in a way suitable for a local-search optimisation algorithm. We note that we need not generally take into account every single point of time within the scheduling window, as the tasks are by definition interruptible at any time without cost. It is easy to understand that the allocation of processors to tasks depends only on the allocation within intervals during which tasks do not change their utility, whereas the allocation at exact points of time is irrelevant due to property no. 5 of value functions. Consider two vectors of local time functions for the same task set with the property that the sum of allocations within intervals without utility change is the same; the two vectors of local time functions can be replaced for each other without changing the system value.

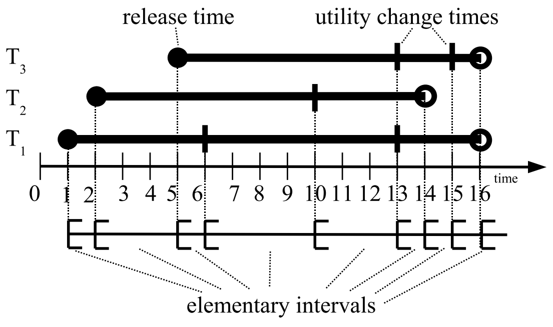
Therefore, we define the search space for the resource allocation algorithm in terms of intervals of time during which the utility does not change; of course, these intervals should be as large as possible to reduce the size of the search space. Keep in mind that trivial intervals of length 1 fulfill the property of no utility changes, but would again result in the same search space as if working with the original local time functions defined for each global time.

First, we define elementary intervals of tasks as the maximum length intervals during which the utility functions of the tasks do not change. These are contiguous subsets of the global time with the lower bound being either the release time or a utility change time (i.e., the utility is smaller than in the immediately preceding time step) and the upper bound being either a utility change time or positive infinity. No utility change time must lie in the interior of such an elementary interval. We do not include intervals prior to the release time of tasks, as we do not allow assignment of computation time to a task in such an interval anyway. Similarly, we do not expect tasks of utility 0 to contribute to the system performance and hence do not include the corresponding time intervals.

For a set of tasks, the set of elementary intervals is defined as the set of maximum length intervals during which none of the tasks change their utility.



a) within interval $[0; \infty[$



b) within interval $[t_0; t_0 + ws[= [1; 16[$

Figure 3.2: Calculation of elementary intervals without utility limit

The calculation of elementary intervals for the example task set is demonstrated in figure 3.2. The sets of utility change times are $\{6, 13\}$ for T_1, $\{10, 14\}$ for T_2, and $\{13, 15\}$ for T_3, so that we receive the following sets of elementary intervals:

$$\begin{aligned} \mathbb{J}_{T_1} &= \{[0; 6[, [6; 13[, [13; \infty[\} \\ \mathbb{J}_{T_2} &= \{[2; 10[, [10; 14[\} \\ \mathbb{J}_{T_3} &= \{[5; 13[, [13; 15[, [15; \infty[\} \end{aligned}$$

Formally, the set of elementary intervals of task T is defined a follows:

Definition 13 (Elementary intervals for task)
We define the set of elementary intervals of task $T \in \mathbb{T}' \subseteq \mathbb{T}$ as

$$\begin{aligned} \mathbb{J}_T := \{\ [t_s; t_e[\subseteq \mathbb{GT} : t_s < t_e \wedge u_T(t_s) > 0 \\ \wedge \quad & (t_s = r_T \vee (t_s > r_T \wedge u_T(t_s - 1) > u_T(t_s))) \\ \wedge \quad & (u_T(t_e - 1) > u_T(t_e) \vee t_e = \infty) \\ \wedge \quad & (\forall t \in \mathbb{GT} : t_s < t < t_e : u_T(t) = u_T(t_s))\} \end{aligned}$$

For the set of tasks, we receive the set of elementary intervals by looking for the least fine-granular intervals that can be mapped into the elementary intervals of individual tasks; in the running example, we get:

$$\mathbb{J}_{\mathbb{T}'} = \{[0;2[, [2;5[, [5;6[, [6;10[, [10;13[, [13;14[, [14;15[, [15;\infty[\}$$

Formally, we calculate the elementary intervals for a task set as:

Definition 14 (Elementary intervals for task set)
We define the set of elementary intervals for task set $\mathbb{T}' \subseteq \mathbb{T}$ *as*

$$\mathbb{J}_{\mathbb{T}'} := \{[t_s;t_e[\subseteq \mathbb{GT} : t_s < t_e \wedge t_s \in PS \wedge t_e \in PS \wedge \forall t : t_s < t < t_e : t \notin PS\}$$

with

$$PS := \{t \in \mathbb{GT} : (\exists T \in \mathbb{T}' \exists [t_s;t_e[\in \mathbb{J}_T : t = t_s \vee t = t_e)\}$$

Note that these definitions have no notion of the scheduling interval previously mentioned other than the task set $\mathbb{T}'$ in general being a proper subset of $\mathbb{T}$, defined by a limited-size time window into the future. The tasks, however, may be ready for execution for an infinitely long time. This does, of course, not imply that the search algorithm within the scheduler would not itself limit its area of interest to a finite time interval (which may be the same as the one used to determine the task set or a smaller one). Modifying the definitions such that intervals are not taken from $\mathbb{GT}$, but from the scheduling window $[t_0; t_0 + ws[$, where t_0 is the current time and ws is the window size, we receive

Definition 15 (Elementary intervals for task within scheduling window)
The set of elementary intervals for task T *in scheduling window* $[t_0; t_0 + ws[$ *is*

$$\begin{aligned}
\mathbb{J}_{T,[t_0;t_0+ws[} \quad := \quad & (\mathbb{J}_T \backslash \{[t_s;t_e[\in \mathbb{J}_T : t_s < t_0 \vee t_e > t_0 + ws\}) \\
& \cup \{[t_0;t_e[: \exists [t_s;t_e[\in \mathbb{J}_T : t_s < t_0 \leq t_e\} \\
& \cup \{[t_s;t_0+ws[: \exists [t_s;t_e[\in \mathbb{J}_T : t_s \leq t_0 + ws < t_e\} \\
& \cup \{[t_0;t_0+ws[: \exists [t_s;t_e[\in \mathbb{J}_T : t_s < t_0 \wedge t_e \geq t_0 + ws\}
\end{aligned}$$

This definition excludes from the original set of intervals those which completely lie outside and truncates those partially lying outside the scheduling interval. For the task set $\mathbb{T}'$, the original definition can be applied on the modified interval sets without change, and we use the notation $\mathbb{J}_{\mathbb{T}',[t_0;t_0+ws[}$ for the set of elementary intervals for task set $\mathbb{T}'$ in the scheduling interval.

Choosing $t_0 = 1$ and $ws = 15$, we now receive to following interval sets for the running example:

$$\begin{aligned}
\mathbb{J}_{T_1,[1;16[} &= \{[1;6[,[6;13[,[13;16[\} \\
\mathbb{J}_{T_2,[1;16[} &= \{[2;10[,[10;14[\} \\
\mathbb{J}_{T_3,[1;16[} &= \{[5;13[,[13;15[,[15;16[\} \\
\mathbb{J}_{\mathbb{T}',[1;16[} &= \{[1;2[,[2;5[,[5;6[,[6;10[,[10;13[,[13;14[,[14;15[,[15;16[\}
\end{aligned}$$

As utility functions are monotonically decreasing, there may obviously be intervals during which the tasks have little or no utility. As the reduction of the number of intervals leads to a smaller search space, it is desirable to focus on those intervals during which significant changes to the system value can be expected. Even though the system value depends on the definition of the value function, tasks with low utility are unlikely to contribute much to the overall value.

Introducing *utility threshold* $\vartheta_u \in \mathbb{R}_0^+$, we can replace the original utility functions by less fine-granular ones obstructing changes in low utility levels by

$$u_T^{\geq\vartheta_u}(t) := \begin{cases} u_T(t) & \text{if } u_T(t) \geq \vartheta_u \\ 0 & \text{if } u_T(t) < \vartheta_u \end{cases}$$

[1]

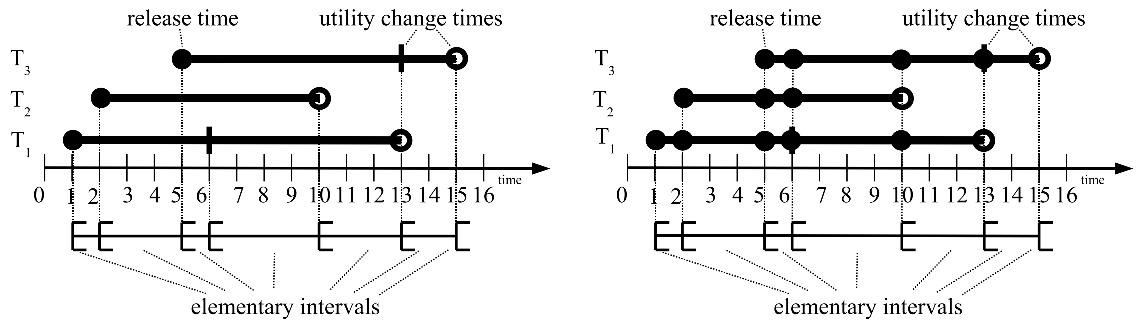
Defining further task $T^{\geq\vartheta_u}$ to have the same properties as T, except for the utility function u_T being replaced by $u_{T^{\geq\vartheta_u}} := u_T^{\geq\vartheta_u}$ and introducing task set $\mathbb{T}'^{\geq\vartheta_u} := \{T^{\geq\vartheta_u} : T \in \mathbb{T}'\}$, we receive sets of high-utility intervals $\mathbb{J}_T^{\geq\vartheta_u} := \mathbb{J}_{T^{\geq\vartheta_u}}$ and $\mathbb{J}_{\mathbb{T}'}^{\geq\vartheta_u} := \mathbb{J}_{\mathbb{T}'^{\geq\vartheta_u}}$ for the original task set $\mathbb{T}'$. Interval sets $\mathbb{J}^{\geq\vartheta_u}_{T,[t_0;t_0+ws[}$ and $\mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$ can be defined as before.

Setting the threshold ϑ_u to 0.3, we get for the example application

$$\begin{aligned}
\mathbb{J}^{\geq 0.3}_{T_1,[1;16[} &= \{[1;6[,[6;13[\} \\
\mathbb{J}^{\geq 0.3}_{T_2,[1;16[} &= \{[2;10[\} \\
\mathbb{J}^{\geq 0.3}_{T_3,[1;16[} &= \{[5;13[,[13;15[\} \\
\mathbb{J}^{\geq 0.3}_{\mathbb{T}',[1;16[} &= \{[1;2[,[2;5[,[5;6[,[6;10[,[10;13[,[13;15[\}
\end{aligned}$$

This threshold is one of the levers for trading off accuracy of scheduling for computational effort. Tasks with utility functions never falling below the given threshold can be considered as

[1]This is of course not the only possibility of reducing the granularity of utility functions; more general heuristics would reduce the number of higher-value utility levels as well.



a) $\mathbb{J}^{\geq 0.3}_{\mathbb{T},[1;16[}$ b) $\mathbb{J}^{\geq 0.3}_{T,[1;16[,\mathbb{T}}$

Figure 3.3: Calculation of elementary intervals with utility limit and task-set-granular intervals

having properties similar to background tasks; their sets of elementary intervals do include an interval with infinite upper bound (not shown in the example).

The final preparatory steps are passing down the granularity of intervals from the task set to individual tasks and determining the set of ready tasks within each interval in $\mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}'}$ with sufficiently high utility.

Definition 16 (High utility elementary intervals at task set granularity)
We define the set of high utility intervals of task $T \in \mathbb{T}' \subseteq \mathbb{T}$ at task set granularity as

$$\mathbb{J}^{\geq\vartheta_u}_{T,\mathbb{T}'} := \{[t_s;t_e[\in \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}'} : t_s \geq r_T \wedge u_T(t_s) \geq \vartheta_u\}$$

Interval sets $\mathbb{J}^{\geq\vartheta_u}_{T,[t_0;t_0+ws[,\mathbb{T}'}$ are defined as before.
In our example, we receive (cf. figure 3.3b))

$$\begin{aligned}
\mathbb{J}^{\geq 0.3}_{T_1,[1;16[,\{T_1,T_2,T_3\}} &= \{[1;2[,[2;5[,[5;6[,[6;10[,[10;13[\} \\
\mathbb{J}^{\geq 0.3}_{T_2,[1;16[,\{T_1,T_2,T_3\}} &= \{[2;5[,[5;6[,[6;10[\} \\
\mathbb{J}^{\geq 0.3}_{T_3,[1;16[,\{T_1,T_2,T_3\}} &= \{[5;6[,[6;10[,[10;13[,[13;15[\}
\end{aligned}$$

Finally, we need to know the set of tasks that may be running with sufficiently high utility in any interval in $\mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$.

Definition 17 (Set of ready tasks)
For an interval $J \in \mathbb{J}^{\geq \vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$, *the set of ready tasks is*

$$\mathbb{T}'_J := \{T \in \mathbb{T}' : J \in \mathbb{J}^{\geq \vartheta_u}_{T,[t_0;t_0+ws[,\mathbb{T}'}\}$$

In the running example, we have

J	$[1;2[$	$[2;5[$	$[5;6[$	$[6;10[$	$[10;13[$	$[13;15[$
$\mathbb{T}'_J$	$\{T_1\}$	$\{T_1, T_2\}$	$\{T_1, T_2, T_3\}$	$\{T_1, T_2, T_3\}$	$\{T_1, T_3\}$	$\{T_3\}$

3.1.2.2 Allocations

Processor time is distributed among tasks according to a matrix of tasks and elementary intervals. Table 3.1 depicts the maximum allocation matrix for the example task set. The entries are the maximum numbers of units of processor time that can be allocated to a task in a specific interval. Obviously, the sum of allocations to all tasks must not exceed the length of the interval. The maximum allocation for a task in an interval equals the length of the interval if the task is ready; otherwise the maximum allocation is 0. We implicitly assume the application has exclusive access to the processor.

	$[1;2[$	$[2;5[$	$[5;6[$	$[6;10[$	$[10;13[$	$[13;15[$
T_1	1	3	1	4	3	0
T_2	0	3	1	4	0	0
T_3	0	0	1	4	3	2
maximum sum	1	3	1	4	3	2

Table 3.1: Allocation constraints table

An allocation of units of computation time to elementary intervals suffices to determine the value, as we can redefine the local time, allocation, and utility functions as well as the value functions, as follows for interval $[t_1; t_2[\in \mathbb{J}^{\geq \vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$:

$$\begin{aligned}
\tau_T : \mathbb{J}^{\geq\vartheta_u}_{T,[t_0;t_0+ws[,\mathbb{T}'} &\to \mathbb{LT}_T \quad \text{with} \quad \tau_T([t_1;t_2[) := \tau_T(t_2-1) \\
\alpha_T : \mathbb{LTF}_T \times \mathbb{J}^{\geq\vartheta_u}_{T,[t_0;t_0+ws[,\mathbb{T}'} &\to \mathbb{LT}_T \quad \text{with} \quad \alpha_T(\tau_T,[t_1;t_2[) := \tau_T(t_2) - \tau_T(t_1) \\
u_T : \mathbb{J}^{\geq\vartheta_u}_{T,[t_0;t_0+ws[,\mathbb{T}'} &\to \mathbb{R}^+_0 \quad \text{with} \quad u_T([t_1;t_2[) := u_T(t_1)
\end{aligned}$$

We noticed earlier that local time and allocation function contain the same information. For evaluating the value function for a resource distribution, local time functions are more appropriate, whereas for the description of search steps, allocation functions are easier. In the following, whenever we talk about allocating cpu time to a task by assigning to the expression $\alpha_T(\tau_T, J)$, we actually intend to set the local time function τ_T such that α_T assumes the desired value. Obviously, allocations to prior intervals must be unambiguously known for this procedure to be well-defined.

The quality function remains unchanged:

$$q_T : \mathbb{LT}_T \to \mathbb{R}^+_0 \quad \text{with} \quad q_T(\tau_T([t_1;t_2[)) := q_T(\tau_T(t_2-1))$$

The value function is defined analogously to the original definition (we do not repeat the examples here); the signature of the value function now is

$$v : \mathbb{QF}_{\mathbb{T}'} \times \mathbb{UF}_{\mathbb{T}'} \times \mathbb{LTF}_{\mathbb{T}'} \times \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[} \to \mathbb{R}^+_0$$

Having established the search space and the value functions to evaluate allocations represented via intervals of time, we can start to search for appropriate allocations to maximise the value function. Well-known local-search techniques can be applied for this purpose. Search steps are changes of the resource distributions within the same intervals. The smallest possible step is to move one unit of processor time from one task to another in one interval and leave all other allocations unchanged. Bigger changes, e.g., moving more than one time unit or modifying the allocations for more than one interval, may be incorporated into one step. Obviously, the selection of a suitable neighbourhood relationship is application-dependent.

The choice of an initial distribution from which to start the optimisation algorithm is another factor influencing the performance. Unfortunately, however, we cannot generally make a better informed choice and usually start with an approximately uniform distribution like the one gained by the algorithm in figure 3.4. Parameters are the set of ready tasks $\mathbb{T}'$ within the current scheduling window and the set of associated high-utility intervals $\mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$.

```
$almostUniform(\mathbb{T}', \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}) : \vec{\tau}$
  forall intervals $J = [t_s; t_e[ \in \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0,t_0+ws[}$ do
    $c := |\mathbb{T}'_J|$        // #active tasks in interval $J$
    $l := t_e - t_s$     // max. allocation in $J$
    $n := 0$
    forall $T \in \mathbb{T}'_J$ do
      $\alpha_T(\tau_T, J) := \lfloor \frac{l}{c} \rfloor + 1 - \min\left(1, \left\lfloor \frac{n}{\max(1, l \mod c)} \right\rfloor\right)$
      $n := n + 1$
    od
  od
  return $\vec{\tau}$
end
```

Figure 3.4: Approximately uniform resource distribution algorithm

	$[1;2[$	$[2;5[$	$[5;6[$	$[6;10[$	$[10;13[$	$[13;15[$
T_1	1	1	0	1	2	0
T_2	0	2	0	1	0	0
T_3	0	0	1	2	1	2
sum	1	3	1	4	3	2

Table 3.2: Approximately uniform distribution for example

Table 3.2 is one possible result for the example task set.[2] Tables 3.3, 3.4, and 3.5 show the local time, quality, and utility for the example task set and the resource allocation in 3.2.

Evaluating this allocation with the value function

$$v_{\vec{q},\vec{u}}(\vec{\tau}, J) = \sum_{T \in \{T_1,T_2,T_3\}} \max_{J' \leq J} u_T(J') \cdot q_T(\tau_T(J')),$$ [3]

we get the overall value of the task set of table 3.6.

[2]depending on the order of tasks in the inner loop

[3]where we define the order relation on time intervals as $[t_s; t_e[\leq [t'_s; t'_e[:\Leftrightarrow t_s \leq t'_s$; as intervals are not overlapping, this even defines a total order on the interval set.

$\tau_T(J)$	$[1;2[$	$[2;5[$	$[5;6[$	$[6;10[$	$[10;13[$	$[13;15[$
T_1	1	2	2	3	5	5
T_2	0	2	2	3	3	3
T_3	0	0	1	3	4	6

Table 3.3: Local time functions for approximately uniform distribution

$q_T(\tau_T(J))$	$[1;2[$	$[2;5[$	$[5;6[$	$[6;10[$	$[10;13[$	$[13;15[$
T_1	0	0	0	0	0.3	0.3
T_2	0	0.1	0.1	0.1	0.1	0.1
T_3	0	0	0	0.4	0.4	0.4

Table 3.4: Qualities for approximately uniform distribution

$u_T(J)$	$[1;2[$	$[2;5[$	$[5;6[$	$[6;10[$	$[10;13[$	$[13;15[$
T_1	1	1	1	0.6	0.6	0
T_2	0	1	1	1	0	0
T_3	0	0	1	1	1	0.7

Table 3.5: Utilities for example task set

$\max_{J' \leq J} u_T(J') \cdot q_T(\tau_T(J'))$	$[1;2[$	$[2;5[$	$[5;6[$	$[6;10[$	$[10;13[$	$[13;15[$
T_1	0	0	0	0	0.18	0.18
T_2	0	0.1	0.1	0.1	0.1	0.1
T_3	0	0	0	0.4	0.4	0.4
$\sum_{T \in \{T_1,T_2,T_3\}} \max_{J' \leq J} u_T(J') \cdot q_T(\tau_T(J'))$	0	0.1	0.1	0.5	0.68	0.68

Table 3.6: Product of quality and utility and system value for approximately uniform distribution

3.1.2.3 Optimisation of Resource Allocation

Starting from the approximately uniform distribution, local-search algorithms are used to improve on this initial solution. From the wide variety of techniques, we implemented a Simulated-Annealing and a Tabu-Search variant to solve the quality-utility scheduling problem.

Having found an appropriate resource allocation for the task set, a schedule can simply be constructed by sorting tasks within each interval arbitrarily. As tasks are independent of each other and context switch costs are not taken into account, the order is irrelevant, as long as the schedule complies with the previously calculated distribution. It may, however, be favourable to prioritise tasks which have already been released or whose release time can be predicted with certainty to reduce the effect of (avoidable) idle phases during the execution of a schedule. This can happen if a task occurs in a schedule which happens not yet to be available at the given time, because the release time estimate was inaccurate. On the same line, ordering tasks with decreasing predictability of release times is an appropriate heuristic for the same objective. One possible schedule for the allocation in table 3.2 is shown in figure 3.5 as a Gantt chart.

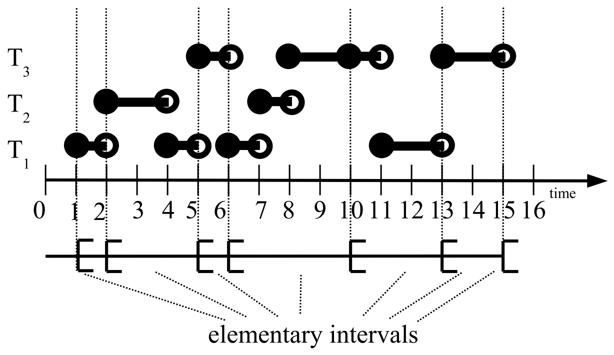


Figure 3.5: Gantt chart for example schedule

3.1.2.3.1 Simulated Annealing

Metropolis et al. described an algorithm for the simulation of a collection of atoms when matter is slowly cooling down [MRR+53]. This so-called Metropolis algorithm is based on the fact that in statistical mechanics, an annealing process consists of first melting the matter at a high temperature and then gradually lowering the temperature until the system freezes and no further changes occur. At each temperature $\mathcal{T}$, enough time must pass in order to allow for the system to reach a steady state. Each configuration of the atoms of the system is defined by the set of atomic positions, $\{p_i\}$, and the energy of the configuration is a function of the atomic positions, $E(\{p_i\})$. The probability of each such configuration is given by the Boltzmann factor $e^{-\frac{E(\{p_i\})}{k_B \cdot \mathcal{T}}}$, where k_B is the Boltzmann constant. Ground (low-energy) states are a very small subset of all configurations; at high temperatures, they are hardly more likely than other states, but they dominate the system at low temperatures. In other words, when the system cools down, it ends up in one of these low-energy states. In practice, cooling must take place very slowly, especially at temperatures close to the freezing point.

Kirkpatrick et al. were probably first in applying the Metropolis algorithms to more general optimisation problems by drawing an analogy between these and the physical process of annealing [KJV83]. The optimisation technique has since become known as *Simulated Annealing*. The first domain Kirkpatrick et al. applied the Simulated Annealing technique to was the physical design of computers, i.e., the placement of elements on a chip and the wiring between them. Identifying the energy of a configuration of atoms with the objective or error function of a combinatorial optimisation problem, one can derive the difference of energy ΔE between two different configurations. Simulated Annealing assumes that given a current state and a candidate successor state, the successor state is accepted

- unconditionally if it is lower in energy (or, in terms of the general optimisation problems, lower in error or higher according to the objective function) *or*
- with probability $e^{-\frac{\Delta E}{k_B \cdot T}}$, if it is higher in energy

One main advantage of Simulated Annealing over iterative refinement algorithms is that it is not bound to get stuck in local optima. At high temperatures, the search is quite likely to leave local optima, because of the high probability of accepting less optimal states. This changes, however, with falling temperature. A second feature is that the gross characteristics of the system already appear at high temperature levels, whereas the more fine-granular details of the solutions develop at lower temperatures.

Our expectation for the Simulated Annealing optimisation scheme is that it is easy to implement and parameterise and yields reasonable results within short computation times. On the other hand, Simulated Annealing does not prevent cyclic search processes, which can constitute a problem especially in low-contrast search spaces, i.e., when values differ only slightly for neighbouring configurations. A further disadvantage of Simulated Annealing is that (in its pure form) it continues to improve on one path of configurations only. No large-distance search steps are taken, rendering the choice of the start configuration an important one. In search spaces with a large number of widely scattered optima, not being able to start new search traces can easily prevent finding solutions close to global optima in finite time.

Figure 3.6 outlines the Simulated-Annealing resource allocation algorithm. Parameters are

- the set of tasks $\mathbb{T}'$
- the set of intervals $\mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$
- the interval with latest start time $J_{max} \in \mathbb{J}^{\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$
- the start temperature for the cooling process $Temp_{start}$

```
$optimize(\mathbb{T}', \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}, J_{max}, Temp_{start}, Temp_{min}, \#rep, cF) : \vec{\tau}_{best}$
  $\vec{\tau} := almostUniform(\mathbb{T}', \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[})$
  $Temp := Temp_{start}$
  $V := v_{\vec{q},\vec{u}}(\vec{\tau}, J_{max})$
  $V_{best} := V$
  $\vec{\tau}_{best} := \vec{\tau}$
  while($Temp > Temp_{min}$) do
    choose one interval $J \in \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$
    for $n = 1$ to $\#rep$ do
      $\vec{\tau'} := searchStep(\vec{\tau})$
      $V' := v_{\vec{q},\vec{u}}(\vec{\tau'}, J_{max})$
      if($V' > V$) then
        $\vec{\tau} := \vec{\tau'}$
        $V := V'$
        if $V > V_{best}$ then
          $V_{best} := V$
          $\vec{\tau}_{best} := \vec{\tau}$
        fi
      else
        with probability $\min(1, e^{\frac{V-V'}{Temp}})$ do
          $\vec{\tau} := \vec{\tau'}$
          $V := V'$
        od
      fi
      $Temp := cF \cdot Temp$
    od
  od
  return $\vec{\tau}_{best}$
end
```

Figure 3.6: Simulated-Annealing resource allocation algorithm

- the minimum temperature for the cooling process $Temp_{end}$
- the number of search steps for a temperature level $\#rep$

- the cool-down factor for the cooling process cF

The return value is the best vector of local time functions found. Several strategies are used to select the interval on which to change the processor time allocation; most of these are based on the number of possible different allocations of resources to tasks for each interval.

3.1.2.3.2 Tabu Search

Tabu Search evolved from Glover's work on integer programming. Although its roots are going back to research in the 1960's, the term itself was coined in an article on the connection between integer programming and artificial intelligence [Glo86]. A good introductory text on Tabu Search can be found in [GTdW93].

Tabu Search is another principle to solve combinatorial and nonlinear problems, the main component being a flexible memory to store previous configurations on the search path, components thereof or operators applied on an initial solution to receive this path. It can be viewed as an iterative technique, repeatedly making moves from one solution to another in the neighbourhood, hoping to gradually find better solutions according to some given objective function. As with Simulated Annealing, the main objective of Tabu Search is to avoid getting stuck in local optima. The problem that can arise in search techniques allowing to proceed with inferior intermediate states is that they can easily run into cycles. Tabu Search tries to explicitly tackle the cyclic search problem by forbidding either to return to states previously encountered or to repeatedly perform the same operations.

One major aspect is to define the set of neighbour states and to select from this neighbourhood the subset of states actually being candidates for a successor state. Scanning the entire neighbourhood is not normally feasible, and it has been noted that the restriction of the candidate states to exactly one element of the neighbourhood means to search according to Simulated Annealing. In general, however, a small set of states in the neighbourhood is selected; this set should be chosen strategically rather than entirely by random, with the goal of increasing the likelihood of ending up with states in the vicinity or direction of a local optimum in the neighbourhood. As a wide variety of heuristics can be applied in order to select a subset of the neighbourhood for each state and these heuristics can themselves be used as search algorithms, Tabu Search has been called a metaheuristic, i.e., a heuristic guiding other heuristics.

Tabu Search stores information that can be used to prevent cycles in the search, either prohibiting or at least penalising moves that would mean returning to a solution previously visited. This information (called *tabu conditions*) is usually structured in one or more *tabu lists*. Basically, there are two possibilities for data to store in tabu lists: either the configurations or states that have recently been visited, or the operators used in the recent past to move from one state to another. In the latter case, it is useful to have reversible operators; this way, it is possible to restore previous states if necessary.

One difficulty with using tabu lists of operators is that they may forbid moves which lead to unvisited and especially attractive unvisited solutions, because the state on which the operator was applied when it was placed in the tabu list can be completely different from the current state. This problem is slightly relieved through the use of several tabu lists for a series of attributes of states, assuming operators to modify only small subsets of these attributes within one move. States having the same settings in these attributes are assumed to behave equally or at least similarly with regard to these attributes. However, this technique is not sufficient. Tabu Search has been extended to incorporate *aspiration level conditions*, which are used to explicitly overrule the tabu conditions. An aspiration level condition is fulfilled if the aspiration level expected when applying an operator on the current state exceeds some threshold value. Moves that would be forbidden according to the tabu specifications are allowed if one or several aspiration level conditions are satisfied.

We expect from the Tabu Search scheme to improve on the Simulated Annealing scheme inasmuch as it prevents at least a large percentage of the cycles in the search process and it includes taking search steps not only in the neighbourhood of the current configuration, but also over larger distances. On the other hand, Tabu Search incurs a larger overhead in both time and space consumption and it is more difficult to implement and parameterise. Without a proper (problem-specific) selection of long-distance search steps and tabu lists, Tabu Search can easily perform worse than Simulated Annealing, especially if compared not only for quality of the resulting schedules, but also on the basis of computational effort to receive these.

The iterative procedure of Tabu Search terminates if either an optimal solution is found, the neighbourhood set of the current state is empty (so that no further move is possible), a maximum number of search steps has been reached, or the state has not changed or noticeably improved for a maximum number of search steps.

Figure 3.7 outlines the Tabu-Search main resource allocation algorithm. We adopted a version of Tabu Search which uses two kinds of search steps: *normal* ones (3.9) making only small modifications to the resource allocation (similar to Simulated Annealing search steps) and *diversification* steps causing radical changes in the allocation (3.8), e.g., as follows:

- choose new allocation completely arbitrarily
- remove all allocations to one task and distribute these resources among the others
- fully allocate the processor to one task in selected intervals

```
$optimize(\mathbb{T}', \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}, J_{max}, maxDiv, lLength, maxImp, impTh) : \vec{\tau}_{best}$
  $\vec{\tau} := almostUniform(\mathbb{T}', \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[})$
  $stop := false$
  $V := v_{\vec{q},\vec{u}}(\vec{\tau}, J_{max})$
  $V_{best} := V$
  $\vec{\tau}_{best} := \vec{\tau}$
  $divCounter := 0; divTrigger := false; improvCounter := 0$
  $tabuList :=< \vec{\tau} >$
  repeat
    if $divTrigger$ then $diversificationSearchStep()$
    else $normalSearchStep()$
    fi
  until stop        // set in $diversificationSearchStep()$ subroutine
  return $\vec{\tau}_{best}$
end
```

Figure 3.7: Tabu-Search resource allocation algorithm

```
$diversificationSearchStep()$
  $divTrigger := false; improvCounter := 0$
  if $divCounter \geq maxDiv$ then $stop := true$
  else
    $divCounter := divCounter + 1$
    $\vec{\tau} := radicalChange(\vec{\tau})$
    $V := v_{\vec{q},\vec{u}}(\vec{\tau}, J_{max})$
    if $V > V_{best}$ then
      $V_{best} := V$
      $\vec{\tau}_{best} := \vec{\tau}$
    fi
  fi
  $tabuList :=< \vec{\tau} >$
end
```

Figure 3.8: Diversification search step in Tabu-Search resource allocation algorithm

```
$normalSearchStep()$
  select an interval $J \in \mathbb{J}^{\geq \vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$
  calculate neighbourhood for $\vec{\tau}$
  $\vec{\tau'} := minorChange(\vec{\tau}, neighbourhood \setminus tabuList)$
  $V' := v_{\vec{q},\vec{u}}(\vec{\tau'}, J_{max})$
  if $tabuList$ has length $lLength$ then
    remove first element from list
  fi
  add $\vec{\tau'}$ to end of $tabuList$
  if $V' > V$ then
    $\vec{\tau} := \vec{\tau'}$
    $V := V'$
    if $(V = 0) \vee (V > 0 \wedge \frac{V'-V}{V} \geq impTh)$ then
      $improvCounter := 0$
    else
      $improvCounter := improvCounter + 1$
    fi
    if $V > V_{best}$ then
      $V_{best} := V$
      $\vec{\tau}_{best} := \vec{\tau}$
    fi
  else
    $improvCounter := improvCounter + 1$
  fi
  if $improvCounter > maxImp$ then
    $divTrigger := true$
  fi
end
```

Figure 3.9: Normal search step in Tabu-Search resource allocation algorithm

As we use the states themselves to define tabu conditions, there is no need for aspiration level conditions.

The parameters for the Tabu-Search resource allocation algorithm are

- the task set $\mathbb{T}'$

- the set of intervals $\mathbb{J}^{\geq \vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$
- the interval with highest start time J_{max}
- the maximum number of diversification steps $maxDiv$
- the maximum length of the tabu list $lLength$
- the maximum number of normal search steps without noticeable improvements before triggering a diversification step $maxImp$
- the threshold for classifying the relative improvement of solutions $impTh$

The return value is the best vector of local time functions found.

3.1.2.4 Search Guidance

Search steps of the local-search algorithms of this section usually incorporate the (random) selection of an interval for which the allocations of time units are moved from one task to another. This applies to both Simulated Annealing search steps and normal search steps in the Tabu Search algorithm. Obviously, it does not make sense to select intervals for which there is no or only one task able to execute, because all possible search steps are trivial. Furthermore, if the number of possible resource distributions within one interval is much larger than in another one, it might be favourable to concentrate on exploring changes occurring in the former rather than in the latter one. We therefore assess the number of possible resource allocations within one interval as follows:

For interval $[t_s;t_e[$, let $\nu := \left|\mathbb{T}'_{[t_s;t_e[}\right|$ be the number of tasks ready for execution within the interval and let $\psi := t_e - t_s$ be the length of the interval (which is, of course, the number of time units that can be distributed among the tasks). Then simple combinatorics (figure 3.10) yields that the local search space for this interval is of size $lss(\psi,\nu)$ with

$$lss(\psi,1) = 1$$

$$lss(\psi,\nu+1) = \sum_{\psi'=0}^{\psi} lss(\psi',\nu)$$

This recursive definition can be resolved as:

$$lss(\psi,\nu) = \begin{pmatrix} \psi+\nu-1 \\ \psi \end{pmatrix}$$

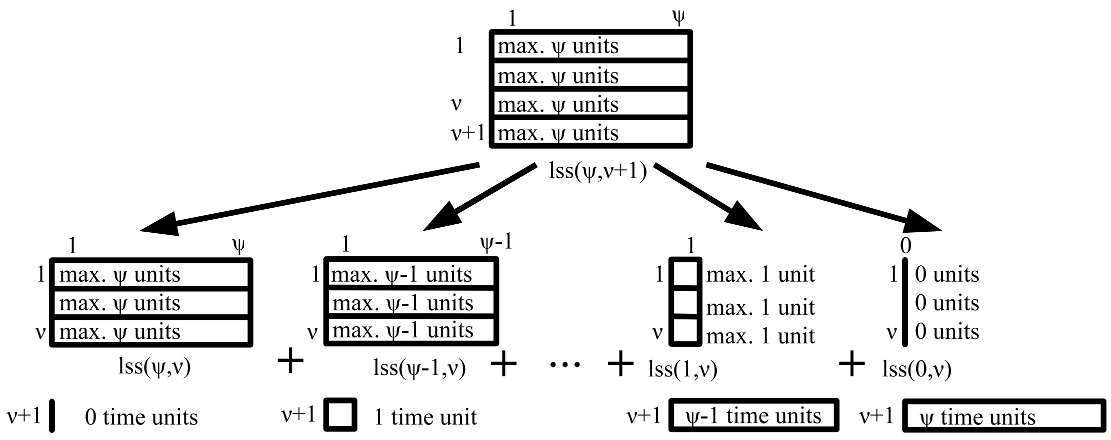


Figure 3.10: Recursive calculation of search space size

Please find the proof in appendix B.6.

Obviously, the size of the search space for a set of tasks $\mathbb{T}'$ in a scheduling window $[t_0; t_0{+}ws[$ of fixed length can be calculated as the product of local search space sizes for the intervals $\mathbb{J}$ into which the scheduling window is broken down. The search space size $ss(\mathbb{T}', \mathbb{J})$ is therefore given as

$$ss(\mathbb{T}', \mathbb{J}) = \prod_{J \in \mathbb{J}} lss\left(|J|, |\mathbb{T}'_J|\right)$$

where $|J|$ is the length of interval J.

The size of the search space obviously depends on the interval set into which the scheduling window is divided. By the definition of the interval borders, there are of course several factors influencing the size of the search space, namely:

- the number of tasks
- the release times of tasks and the regularity of release (e.g., periodic tasks)
- for periodic task sets (i.e., all tasks are released periodically), the homogeneity of the period lengths and the phase shifts
- the number of utility levels for each task

No general rule is given here for all the factors as to how they influence the size of the search space, but there are some important cases which deserve special treatment. Especially for the very common case of periodic task sets, the number of tasks being released during any period of time (in this case: during the scheduling window) can be estimated by dividing the number of tasks being released within the least common multiple of all period lengths of tasks by the length of this time interval. This *average release frequency* is a good estimate for the size of the search space for harmonic periodic task sets[4]. In the general case, however, the average release frequency is neither easy to determine nor a good estimate for the size of the search space.

Therefore, we instead calculate the expected release times for the (finite) task set within the scheduling window, decompose the scheduling window into an appropriate set of intervals and explicitly calculate both the size of the local search space for each interval and the size of the global search space for the entire scheduling window. The logarithm of the local search space size to the basis of the global search space size is then used as weight for the selection of intervals in the search process:

$$sProb_J := \log_{ss(\mathbb{T}',\mathbb{J})} lss\left(\left|J\right|, \left|\mathbb{T}'_J\right|\right)$$

This means that within one search step, changes in interval J_1 are twice as likely than changes in interval J_2 if $sProb_{J_1} = 2 \cdot sProb_{J_2}$.

We thus direct computational effort towards those intervals with big local search spaces, as these are now more likely to be selected within a search step.

Before starting the local search algorithm, the selection of an interval set for a given scheduling window is obviously very important. It can be rewarding to consider reducing the number of elementary intervals, e.g., by modifying the release time estimates for some tasks. The size of the search space may shrink significantly due to the reduced interval set and allow the scheduling algorithm to make up for the approximation of information on the task set incurred by the modifications.

The search space is biggest (for a given number of intervals and a given number of ready tasks in these intervals) if the intervals have equal length. Again, the proof can be found in the appendix (B.7). If the number of intervals cannot be reduced without losing too much information, it may therefore also be possible to reduce the size of the search space by intentionally shifting the borders of adjacent intervals and explore possibly invalid good solutions in the vicinity of valid solutions. In many cases, the search space is well-formed enough to derive good valid solutions from the invalid ones gained by this strategy.

[4]i.e., each period length is either the multiple of another one ore vice versa

3.1.3 Lagrange Multiplier Approach

Lagrange multipliers are a widely used method for finding extrema on a bounded surface. This optimisation technique goes back to the “calculus of variations” of Joseph Louis Lagrange (1736-1813).

Suppose we want to find the extremum of a continuously differentiable function

$$f : \mathbb{R}^n \to \mathbb{R}$$

subject to a constraint

$$g(x_1, \ldots, x_n) = C \in \mathbb{R}$$

where

$$g : \mathbb{R}^n \to \mathbb{R}$$

is continuously differentiable.

Define the set of points satisfying the constraint as

$$M = \{(x_1, \ldots, x_n \in \mathbb{R}^n | g(x_1, \ldots, x_n) = C\}.$$

If $(x_1, \ldots, x_n)$ is a local extremum of $f|_M$, then the gradients grad $f(x_1, \ldots, x_n)$ and grad $g(x_1, \ldots, x_n)$ are linearly dependent, i.e., there is a $\lambda \in \mathbb{R}$ such that

$$\text{grad } f(x_1, \ldots, x_n) = \lambda \cdot \text{grad } g(x_1, \ldots, x_n)$$

or

$$\text{grad } g(x_1, \ldots, x_n) = \lambda \cdot \text{grad } f(x_1, \ldots, x_n)$$

The factor λ is called the *Lagrange multiplier*. For a proof, see [**?**].

Cheng [Che02] suggests to apply the method of Lagrange multipliers to the time-budgeting problem (which is similar, albeit simpler than the quality/utility problem, because the search space is time-invariant). The first step is to approximate the given discrete functions by continuously differentiable functions as required by the Lagrange multiplier method.

We applied the method only for value functions based on the sum of the products of quality and utility functions without maximum operator; in this case, the sum of approximations of functions for individual tasks can be used as approximations for the sum of functions.

At time instant t_0, for task T form the product of quality and utility as:

$$qu_T(t_T) := u_T(t_T) \cdot q_T(\tau_T(t_0) + t_T - t_0)$$

where we evaluate the quality function according to the units of processor time actually allocated in the past and the maximum processor allocation in the future. It is the aim of the optimisation process to reduce the allocation to each task in order to be able to meet the (resource) constraint of only one processor to be shared among the tasks.

We approximate each discrete function qu_T defined by data points $(t_{T,1}, qu_T(t_{T,1})), \ldots, (t_{T,n_T}, qu_T(t_{T,n_T}))$ by a quadratic function $\widehat{qu}_T(t_T) = a_{T,0} + a_{T,1}t_T + a_{T,2}t_T^2$ using the least-squares method:

$$\begin{pmatrix} a_{T,0} \\ a_{T,1} \\ a_{T,2} \end{pmatrix} = \left(\begin{pmatrix} 1 & 1 & \ldots & 1 \\ t_{T,1} & t_{T,2} & \ldots & t_{T,n_T} \\ t_{T,1}^2 & t_{T,2}^2 & \ldots & t_{T,n_T}^2 \end{pmatrix} \begin{pmatrix} 1 & t_{T,1} & t_{T,1}^2 \\ 1 & t_{T,2} & t_{T,2}^2 \\ \vdots & \vdots & \vdots \\ 1 & t_{T,n_T} & t_{T,n_T}^2 \end{pmatrix} \right)^{-1} \cdot$$

$$\cdot \begin{pmatrix} 1 & 1 & \ldots & 1 \\ t_{T,1} & t_{T,2} & \ldots & t_{T,n_T} \\ t_{T,1}^2 & t_{T,2}^2 & \ldots & t_{T,n_T}^2 \end{pmatrix} \begin{pmatrix} qu_T(t_{T,1}) \\ qu_T(t_{T,2}) \\ \vdots \\ qu_T(t_{T,n_T}) \end{pmatrix}$$

According to the description above, we receive a set of linear equations as the partial derivatives of these quadratic functions:

$$\begin{aligned} 2a_{T_1,2}t_{T_1} + a_{T_1,1} &= \lambda \\ 2a_{T_2,2}t_{T_2} + a_{T_2,1} &= \lambda \\ &\vdots \\ 2a_{T_n,2}t_{T_n} + a_{T_n,1} &= \lambda \end{aligned}$$

Adding the resource constraint

$$t_{T_1} + t_{T_2} + \cdots + t_{T_n} = ws$$

with ws being the size of the scheduling window, we have a system of $n+1$ linear equations in $n+1$ variables, which can easily be solved. Taking care when handling quadratic curves with the same derivative and rounding solutions to the nearest integer numbers, the Lagrange method can be used to solve the quality/utility scheduling problem. However, it must be noted that it does not provide optimal solutions in this case, as the product of quality and utility function is not time-invariant, i.e., it is valid only for the next time step; for larger window sizes, optimisation is performed on a mere estimate of the future problem space. Furthermore, these functions are themselves approximated, and, finally, the solutions have to be rounded to integer numbers. [5]

[5] As an alternative, we also approximated the given discrete function by quadratic splines in the implementation of the scheduling algorithm; we will not describe this alternative in this work, because it would not add anything in principle to the discussion.

Our expectations in the Lagrange multiplier approach are limited. Although known to be an effective means for a wide variety of optimisation problems, we face several sources of approximation necessary to find a formulation of our problem suitable for Lagrangian optimisation. The original quality and utility functions have to be approximated by continuously differentiable functions. Depending on the type of approximation, these can be arbitrarily bad models for the originals. The optimisation scheme suffers from the fact that the search space for the given problem class constitutes a mere estimate for the value which can be obtained in the future (expressed by the notion of upper bounds of value in the preceding chapter). This estimate becomes increasingly inaccurate with the distance of the corresponding time from current time. Furthermore, our Lagrangian-optimisation-based scheduler can deal only with a restricted set of problems, i.e., with non-hierarchical task sets without precedence constraints, and with exactly one kind of value function. Nevertheless, this scheme has been included for comparison primarily because it constitutes a well-known standard optimisation technique and has even been previously applied successfully to the time budgeting problem, which is related to our problem class.

3.2 Proactive Conditional Scheduling

Another solution technique we considered for solving the quality / utility scheduling problem was by formulating it as a Markov Decision Process (MDP) and deriving appropriate execution policies for each time instant.

3.2.1 Description

MDPs consist of a stochastic automaton on a finite set of world states $\mathbb{S}$ and a finite set of actions $\mathbb{A}$. Probabilities are defined for transitions of one state s to another one s' when performing action a. For each state, the current policy π determines which action to take next. Therefore, instead of calculating schedules explicitly, the goal of our MDP-based scheduler is to calculate appropriate policies.

First, let us define a set of possible states for every point of time:

$$\mathbb{S}_t = \{(t, \langle a_0, \ldots, a_t \rangle, \mathbb{T}(t)\}$$

with

$$t \in \mathbb{GT}\text{ , } \mathbb{T}(t) \text{ being the tasks released up to } t,$$

the history of actions until time t

$$a_0, \ldots, a_t \in \mathbb{T}'$$

and the set of states as

$$\mathbb{S} = \bigcup_{t \in \mathbb{GT}} \mathbb{S}_t.$$

A state includes all allocations of processing time to tasks up to a certain time t and the set of ready tasks at this time.

For the time being, it suffices to define the set of actions as tasks being executed:

$$\mathbb{A} :\equiv \mathbb{T}'$$

An action is made up of the allocation of processing time to all tasks at a specific time.

Note that the sets of states and actions are infinite, in contrary to the above stated requirements. Therefore, instead of working on the entire sets, we use finite and mutable subsets $\hat{\mathbb{S}}$ (the *state envelope*) and $\hat{\mathbb{A}}$ (the *action envelope*). To represent all states not present in the envelope, we add two pseudo states s_{out} (to be used for legal transitions) and s_{err} (to be used for illegal transitions). Finally, we need an additional start state s_0.

To determine the probability for transitions between states, first take into account that it is neither possible to skip a point in time while progressing in producing and executing a schedule, nor to go back in time. Furthermore, the action performed in any step has to be incorporated into the next state. Consider two states

$$s = (t, \langle a_0, \dots, a_t \rangle, \mathbb{T}(t))$$
$$s' = (t', \langle a'_0, \dots, a'_{t'} \rangle, \mathbb{T}(t')).$$

Let $p_T(t)$ denote the probability of task T being released at time t and

$$\Delta_T(t+1) := \begin{cases} 1 & \text{if } T \in \mathbb{T}(t+1) \backslash \mathbb{T}(t) \\ 0 & \text{otherwise} \end{cases}$$

The probability $Pr(s, a, s')$ of going to state $s' \in \hat{\mathbb{S}}$ from state $s \in \hat{\mathbb{S}}$ when executing action a, is

$$Pr(s, a, s') = \begin{cases} \prod_{T \in \mathbb{T}'} (p_T(t+1))^{\Delta_T(t+1)} \cdot (1 - p_T(t+1))^{1-\Delta_T(t+1)} & \\ \qquad \text{if } t' = t+1 \wedge a'_{t+1} = a \wedge \forall_{t' \in \{0,\dots,t\}} : a_{t'} = a'_{t'} & \\ \qquad \wedge \langle a'_0, \dots, a'_{t'} \rangle \text{ is a valid partial schedule} & \\ 0 & \text{otherwise} \end{cases}$$

$$Pr(s, a, s_{out}) = \begin{cases} 1 - \sum_{s' \in \hat{\mathbb{S}}} Pr(s, a, s') & \\ \qquad \text{if } t' = t+1 \wedge a'_{t+1} = a \wedge \forall_{t' \in \{0,\dots,t\}} : a_{t'} = a'_{t'} & \\ \qquad \wedge \langle a'_0, \dots, a'_{t'} \rangle \text{is a valid partial schedule} & \\ 0 & \text{otherwise} \end{cases}$$

$$Pr(s, a, s_{err}) = \begin{cases} 1 & \text{if } \exists T \in \mathbb{T}' : r_T > t \land \exists t' \leq t : a_{t'} = T \\ 0 & \text{otherwise} \end{cases}$$

$$Pr(s_{out}, a, s) = Pr(s_{err}, a, s) = 0$$

$$Pr(s_{out}, a, s_{out}) = Pr(s_{err}, a, s_{err}) = 1$$

An instantaneous reward is assigned to each state to indicate the gain a certain allocation of processing time to tasks means for the performance of an application. The reward is derived from the value function of the task set. Define

$$\vec{\tau}_{a_0,\ldots,a_t}$$

such that

$$\alpha_T(\tau_{T,a_0,\ldots,a_t}, t') = \begin{cases} 1 & \text{if } a_{t'} = T, 0 \leq t' \leq t \\ 0 & \text{otherwise} \end{cases}$$

Then the reward is the difference of objective function evaluations for consecutive allocations, i.e., $R : \hat{\mathbb{S}} \to \mathbb{R}$ as follows:

$$R(s) := \begin{cases} v_{\vec{q},\vec{u}}(\vec{\tau}_{a_0,\ldots,a_t}, t) - v_{\vec{q},\vec{u}}(\vec{\tau}_{a_0,\ldots,a_t}, t-1) & \text{if } t > 0 \\ v_{\vec{q},\vec{u}}(\vec{\tau}_{a_0,\ldots,a_t}, t) & \text{if } t = 0 \end{cases}$$

The goal is to find an appropriate policy $\pi : \hat{\mathbb{S}} \cup \{s_0, s_{out}, s_{err}\} \to \hat{\mathbb{A}}$ for the system to know how to act in any situation.

To assess a state s when encountered under the application of policy π, a discounted sum of the expected future reward is used:

$$\begin{aligned} V_\pi(s) &= \sum_{t=0}^{\infty} \gamma^t E(R_t) \\ &= R(s) + \gamma \sum_{s' \in \hat{\mathbb{S}}} Pr(s, \pi(s), s') V_\pi(s') \end{aligned} \tag{3.1}$$

where R_t is a random variable for the reward at time t and the discount factor $\gamma \in [0; 1[$ determines the influence of future rewards on current decisions. A discount factor of 0 means that decisions are solely based on immediate rewards, a discount factor close to 1 allows distant future behaviour to have considerable effect on the current policy.

A policy π is considered superior to a policy π', if

- $\forall s \in \hat{\mathbb{S}} : V_\pi(s) \geq V_{\pi'}(s)$
- $\exists s \in \hat{\mathbb{S}} : V_\pi(s) > V_{\pi'}(s)$

We use a policy iteration algorithm to determine an optimal policy for the current envelope (figure 3.11).

```
policy_iteration()
  let $\pi'$ be an arbitrary policy on $\hat{\mathbb{S}}$
  repeat
    $\pi := \pi'$
    forall $s \in \hat{\mathbb{S}}$ do
      calculate $V_\pi(s)$ by solving system of linear equations (3.1)
    od
    forall $s \in \hat{\mathbb{S}}$ do
      forall $a \in \hat{\mathbb{A}}$ do
        if $R(s) + \gamma \sum_{s' \in \hat{\mathbb{S}}} Pr(s,a,s')V_\pi(s') > V_\pi(s)$ then $\pi'(s) := a$
        else $\pi'(s) := \pi(s)$
      od
    od
  until $\pi = \pi'$
  return $\pi$
end
```

Figure 3.11: Policy iteration algorithm

We interleave calculation of policies with their application and the execution of application task instances in a so-called recurrent deliberation model (3.12). As the scheduling algorithm uses the same computing resources as the application tasks, we have to operate on the tradeoff between scheduling effort and the quality of the resulting policy. The quality of a policy can be measured by the size of the envelope, as it is desirable to minimise the number of states for which the system has to act according to the default actions. State space pruning is done by removing those from the set of active tasks whose utility has fallen below a predefined utility threshold.

```
recurrent_deliberation()
  $s := s_0$
  forever do
    while scheduling allowance not reached do
      extend the envelope $\hat{\mathbb{S}}$ by forward simulation
      prune the envelope $\hat{\mathbb{S}}$
      generate an optimal policy $\pi$ for $\hat{\mathbb{S}}$
    od
    while application task allowance not reached do
      execute application tasks according to policy $\pi$
    od
  od
end
```

Figure 3.12: Recurrent deliberation algorithm

3.3 Experimental Data

All of the scheduling schemes described in the preceding sections have been implemented within a simulation environment for scheduling problems. Details of this environment as well as experimental results from a series of benchmark tests can be found in the following chapters.

Chapter 4

General Quality / Utility Scheduling Problem

Even if there is only one possible unified theory, it is just a set of rules and equations. What is it that breathes fire into the equations and makes a universe for them to describe? The usual approach of science of constructing a mathematical model cannot answer the questions of why there should be a universe for the model to describe. Why does the universe go to all the bother of existing?

Stephen Hawking
[A Brief History of Time]

Chapter 2 introduced the basic quality / utility scheduling problem on a single-processor system for unstructured task sets in order to provide the reader with a smooth approach and to allow a simple description of the scheduling algorithms, concentrating on the main characteristics of the problem, i.e., the correlation between task-local properties on the one hand and global time attributes on the other hand. In our experimental work, however, we use a more general model than this one. In this chapter, we will provide a complete coverage of this extended model for quality / utility scheduling before proceeding to details on experimental results in subsequent chapters.

4.1 Extended Model for Quality / Utility Scheduling Problems

The model for problems as introduced earlier consisted of an unstructured set of tasks and a single processor (figure 4.1). The main attributes of tasks were the release time and quality and utility functions.

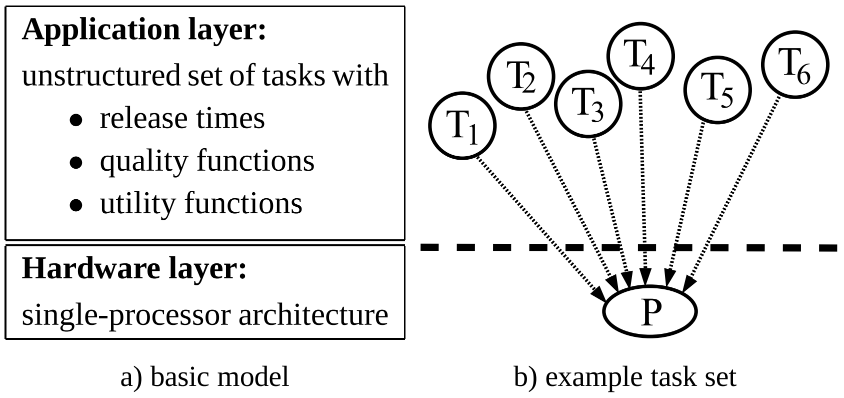


Figure 4.1: Basic model for the quality / utility scheduling problem

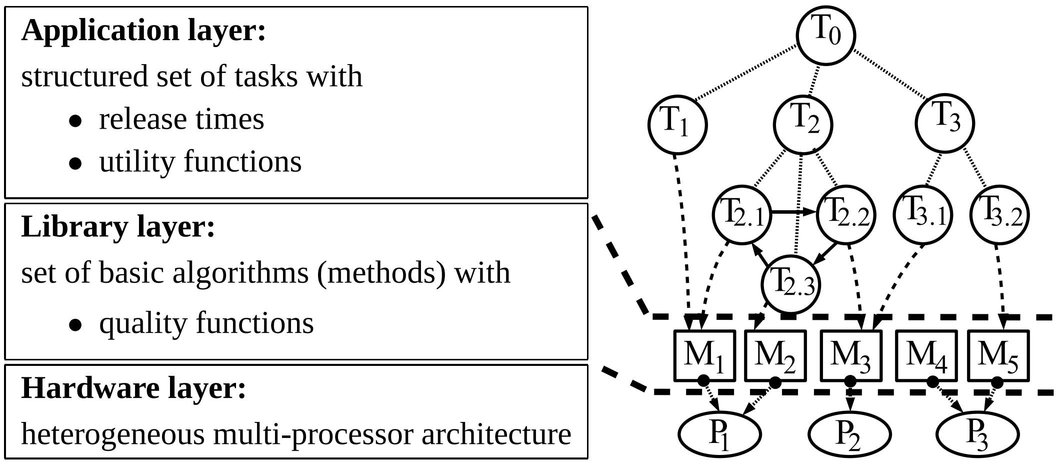
In the general version, the task set is structured in a way that will be explained in detail later in this chapter. For the moment, note the two different graph structures defined on the nodes $T_2, T_{2.1}, T_{2.2}$, and $T_{2.3}$ of the example graph in figure 4.2b). Tasks $T_{2.1}, T_{2.2}$, and $T_{2.3}$ are subtasks of T_2 and form a hierarchy graph together with their common parent. A so-called dependency graph is spanned by the tasks $T_{2.1}, T_{2.2}$, and $T_{2.3}$, representing a certain form of precedence constraints. Furthermore, a set of basic algorithms is introduced in between the task and hardware layers to facilitate the reusability of frequently needed computations. Finally, instead of a single processor architecture, we now target a heterogeneous multiprocessor architecture.

Basic computations are provided to the application developer as a method library, on top of which he or she can construct an application as a hierarchy of tasks.

In the following sections, the general model is gradually developed from the basic scheme.

4.2 Processors, Methods and Tasks

In this section, our original model is extended in various directions. First, we now target a heterogeneous multiprocessor system rather than a single processor. Furthermore, we split the quality and timeliness flexible aspects and place them on distinct objects, based on the perception that quality profiles are often associated with basic algorithms that can be reused in later applications



a) general model b) example application graph

Figure 4.2: General model for the quality / utility scheduling problem

once implemented. A set of methods is used to represent such a library of reusable basic algorithms; the quality functions can be stored within such a library, whereas utility functions are problem-specific and must be stored separately for each application. Finally, the concept of task and method instances is introduced to increase the practical usability of the model by providing a simple means of expressing infinite sets of computations by finite sets of task and method specifications.

4.2.1 Hardware Layer

We assume that the hardware layer of the general model consists of a heterogeneous multiprocessor architecture. For simplicity, we assume further that no other resources are considered in our model and that context switch, migration, and communication costs are neglected. Therefore, we do not target a specific topology, but rather assume the communication between any two processors to be ideal (loss-free and without delay). We introduce the (finite) set $\mathbb{P} = \{P_1, \ldots, P_{|\mathbb{P}|}\}$ of all processors.

4.2.2 Library Layer

Methods can be thought of as basic algorithms available to the systems engineer as an algorithm library. The algorithms can be instantiated in an arbitrary number and associated with tasks to build an application. We suppose methods (and hence, the tasks based on them) to be interruptible at any time. Methods[1] do not lose data when they are interrupted and can resume operation on the same processor immediately and without need for rollbacks or redo mechanisms once the interrupt has finished. The quality function tells the environment the value to be expected from (an instance of) a method when assigned a certain amount of processor time.

We denote the (finite) set of methods by $\mathbb{M} = \{M_1, \ldots, M_{|\mathbb{M}|}\}$.

Definition 18 (Method instances)
$I_M^1, I_M^2, \ldots$ *are the instances of* $M \in \mathbb{M}$.
$\mathbb{I}_M := \{I_M^1, I_M^2, \ldots\}$ *is the set of all instances of* M.
$\mathbb{I}_{\mathbb{M}'} := \bigcup_{M \in \mathbb{M}'} \mathbb{I}_M$ *is the set of all instances of methods in* $\mathbb{M}' \subseteq \mathbb{M}$.

Methods may in general only be executable on a subset of all processors, and their execution speed may vary, e.g., according to processor speed or dedicated support of specific arithmetic operations (e.g., floating-point calculations). To express the progress of a method depending on the processing time allocated to it, quality functions are introduced. Remember that in the basic scheme, quality functions were associated directly with tasks. We assume quality functions for methods to be known in advance and to remain unchanged regardless for which task the computations are carried out. These quality functions are attributes of the methods and are stored statically with the method library.

First, we need to extend the concept of local time introduced previously. The local time of a method with respect to a processor is the amount of computation time on this processor awarded to the method.

Definition 19 (Local time for method instances)
The set of local times of method instance $I \in \mathbb{I}_M$, $M \in \mathbb{M}$ *with respect to processor* $P \in \mathbb{P}$ *is written* $\mathbb{LT}_{P,I} :\equiv \mathbb{N}_0$.

Quality functions map the allocation of computation time on a processor to the progress of the method (instance) expressed through its quality.

Definition 20 (Quality functions)
Every method $M \in \mathbb{M}$ *has a time-discrete monotonically increasing function (the quality function) for each processor* $P \in \mathbb{P}$.

$$q_{P,M} : \bigcup_{k \in \mathbb{N}} \mathbb{LT}_{P,I_M^k} \to \mathbb{R}_0^+ \qquad \textit{with} \qquad q_{P,M}(0) = 0 \qquad \textit{and}$$

$$q_{P,M}(n) = q_{P,M}(n') \textit{ for all } n \in \mathbb{LT}_{I_M^k}, n' \in \mathbb{LT}_{I_M^{k'}}, n \equiv n' \textit{ and } k, k' \in \mathbb{N}$$

[1]more precisely: method instances, as we will see shortly

Awarding the same processing time on the same processor to instances of the same method yields the same quality.

As before, we introduce vectors of quality functions for notational brevity.

Definition 21 (Vectors of quality functions)
The set of vectors of all possible quality functions for the elements of a method set $\mathbb{M}' \subseteq \mathbb{M}$ *is*

$$\mathbb{QF}_{\mathbb{M}'} := \prod_{M \in \mathbb{M}', P \in \mathbb{P}} \mathbb{QF}_{M,P} \qquad \textit{with} \qquad \mathbb{QF}_{M,P} = (\mathbb{R}_0^+)^{\bigcup_{k \in \mathbb{N}} \mathbb{LT}_{P,I_M^k}}.$$

We use the notation $\vec{q} \in \mathbb{QF}_{\mathbb{M}'}$ *for vectors of quality functions for all methods in* $\mathbb{M}$*.*

Local time functions still represent the number of units of processor time that have been awarded to a method up to a certain time. However, we must now parameterise local time functions, as we target a multiprocessor system.

Definition 22 (Local time and allocation functions for method instances)
For all method instances $I \in \mathbb{I}_{\mathbb{M}}$*, we define a local time function with regard to a certain processor* $P \in \mathbb{P}$ *as a monotonically increasing function*

$$\tau_{P,I} : \mathbb{GT} \to \mathbb{LT}_{P,I} \textit{ with } \forall t \leq r_I : \tau_{P,I}(t) = 0$$

The set of all possible local time functions for I *on processor* $P \in \mathbb{P}$ *is* $\mathbb{LTF}_{P,I}$*.*
Allocation functions

$$\alpha_{P,I} : \mathbb{LTF}_{P,I} \times \mathbb{GT} \to \mathbb{LT}_{P,I}$$

are introduced the same way as before:

$$\alpha_{P,I}(\tau_{P,I}, t) := \tau_{P,I}(t+1) - \tau_{P,I}(t)$$

Definition 23 (Vectors of local time functions for method instances)
For a subset $\mathbb{I}' \subseteq \mathbb{I}_{\mathbb{M}}$ *of method instances,* $\mathbb{LTF}_{\mathbb{I}'} := \prod_{P \in \mathbb{P}, I \in \mathbb{I}'} \mathbb{LTF}_{P,I}$ *is the set of vectors of local time functions for method instances in* $\mathbb{I}'$*; we use the symbol* $\vec{\tau} \in \mathbb{LTF}_{\mathbb{I}'}$ *for elements of this set.*

To prevent migration of method instances between different processors, we claim that

$$(\exists I \in \mathbb{I}_M : \exists t \in \mathbb{GT} : \tau_{P',I}(t) > 0 \wedge \tau_{P,I}(t) > 0) \Rightarrow P = P'$$

Finally, vectors $\vec{\tau} \in \mathbb{LTF}_{\mathbb{I}'}$ of local time functions are still bounded by the same resource constraints as before:

$$\forall P \in \mathbb{P} : \forall t \in \mathbb{GT} : \sum_{I \in \mathbb{I}'} \alpha_{P,I}(\tau_{P,I}, t) \leq 1$$

4.2.3 Application Layer

We note that in many applications, significant subsets of the task set behave periodically, i.e., the tasks are released in regular intervals. For other task sets, the release times may not be equidistant, but distributed with a known function. It is generally beneficial for a scheduling algorithm to use this additional information on the regularity of release times. For this purpose, we group together tasks correlated this way. Such tasks have previously been treated as entirely independent entities and will now be considered as one task with several *instances*. Tasks are associated with methods to express that the task can be implemented by these basic algorithms. Typically, a task can be implemented by a set of different algorithms with specific resource requirements, execution time, accuracy, level of detail, etc. In our model, we implement this method selection scheme by a number of methods for each task; these methods in general have different quality functions.

Definition 24 (Child function for tasks)
Each task $T \in \mathbb{T}$ is associated a non-empty set of methods. This relationship can be expressed by a graph structure on the tasks and methods spanned by a child function

$$c : \mathbb{T} \rightarrow 2^{\mathbb{M}}$$

The child function crosses the border between application and library layer.
We now assume the task set $\mathbb{T}$ to be finite, and for each task, instances are released.

Definition 25 (Task instances)
For task $T \in \mathbb{T}$, $\mathbb{I}_T = \{I_T^1, I_T^2, \dots\}$ is the set of instances of T.
The instance set of a task is either infinite or has exactly one element. We do not consider the case that a finite number of instances greater than 1 is released.
The set of all instances of all tasks in $\mathbb{T}$ is

$$\mathbb{I}_{\mathbb{T}} := \bigcup_{T \in \mathbb{T}} \mathbb{I}_T$$

Note that the task set is now assumed to be finite, so that a finite set of task instances within a scheduling window can be defined as the union of the respective sets of instances of each task, denoted by $\mathbb{I}'_{\mathbb{T}} \subseteq \mathbb{I}_{\mathbb{T}}$.

The child functions defined above between tasks and methods are now extended to task and method instances.

Definition 26 (Child function for task instances)
Each task instance I_T^k is associated with exactly one instance $I_M^{k'}$ for each method $M \in c(T)$. The child relationship between tasks and methods is extended to the instances:

$$c : \mathbb{I}_{\mathbb{T}} \to 2^{\mathbb{I}_{\mathbb{M}}}$$

The instance numbers of task instances and their associated method instances are not necessarily equal. This is unavoidable if we want instance numbers to be unique within all the instances of a task or method on the one hand and allow the same method to be used by several tasks on the other. For example, let $M \in c(T_1) \cap c(T_2)$. The requirement of equal instance numbers for task and method instances would leave us with the question of associating the $k - th$ instance of M, I_M^k, with either $I_{T_1}^k$, $I_{T_2}^k$, or both. Associating I_M^k with both task instances would mean the task instances sharing the same invocation of the same piece of code with the same parameters and data, an assumption which is not valid in most cases. Of course, we cannot associate the method instance with only one of the task instances, as the other one would be missing an implementation alternative (which might even be the only one).

We therefore pose that for any two task instances $I, I' \in \mathbb{I}_{\mathbb{T}}$ with $I \neq I'$, $c(I) \cap c(I') = \emptyset$. This definition explicitly prevents two task instances to share the same method instance, regardless of whether they are instances of the same method or not.

Method instances are created only when needed, i.e.,

$$\forall I_M \in \mathbb{I}_{\mathbb{M}} : \exists I_T \in \mathbb{I}_{\mathbb{T}} : I_M \in c(I_T)$$

We intentionally overloaded the symbol c for the child functions on task/method and task/method instance levels to express the close relationship between them.

Similarly to the prior definition for tasks, release times are now associated with task instances. Method instances inherit the release times of the task instances which they implement.

Definition 27 (Release times)
The release time for task instance $I \in \mathbb{I}_{\mathbb{T}}$ is $r_I \in \mathbb{GT}$.
A method instance has the same release time as the unique task instance it is associated with:

$$r_{I_M^k} := r_{I_T^{k'}} \quad \textit{with} \quad I_M^k \in c(I_T^{k'})$$

Of course, release times of task instances can be arbitrary, and it is not always easy to find a mathematical description for the release behaviour of a task set in reality. Within this work, however, we assume that the release times of the instances of a task can be fully described by a stochastic distribution function. We even constrain our model to two kinds of functions which we found both easy to describe and sufficiently close to many real-world scenarios. Further extensions in this direction would probably not add any more insights in the nature of our scheduling problem. Therefore, we distinguish only three types of tasks with regard to their release behaviour.

Periodic tasks have instances that are released at (approximately) equidistant times, the distance being called the *period length*; actual release times may deviate from the beginning of a task period only within very small limits. This deviation is called the *maximum release time jitter*. We assume the release time to be distributed uniformly around the beginning of a task period; the task periods may have a constant offset from the start time of the system (global time 0) called the *phase shift*. Release times of instances of periodic tasks are not correlated, i.e., the release time distribution of one instance is independent of the release time of preceding instances. Specifically, if the jitter was allowed to be big enough compared to the period length, task instances might "overtake" each other, so that instances with higher instance numbers are released earlier than instances with lower instance numbers. Such a situation is generally considered undesirable and can be guaranteed not to occur if the maximum jitter is sufficiently small compared to the period length. To sum up, the attributes of a periodic task T (see figure 4.3) are

- period length per_T
- phase shift φ_T
- maximum jitter j_T

The release time distribution is

$$Pr(r_{I_T^k} = t) = \begin{cases} \frac{1}{2j_T+1} & \text{if } \varphi_T + (k-1)per_T - j_T \leq t \leq \varphi_T + (k-1)per_T + j_T \\ 0 & \text{otherwise} \end{cases}$$

Aperiodic tasks have instances that are released with the same probability at any time instant. This *release probability* defines a geometric distribution of release times, which basically means that the probability of a new task instance being released remains constant over time[2]. Release times of consecutive instances differ by at least a *minimum interarrival time*. Once the minimum interarrival time has passed, a new instance is released with a constant release probability p_T

[2]The geometric distribution is the equivalent of the memoryless exponential distribution in discrete domains.

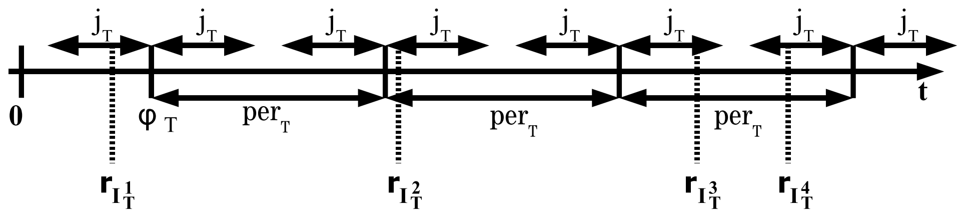


Figure 4.3: Instantiation of periodic task

at each time step. Release times of instances of aperiodic tasks are correlated, i.e., the release time distribution of one instance is dependent on the release time of the immediately preceding instance. The expected value of the release time after the minimum interarrival time has passed is $\frac{1}{p_T}$. The release time of the first instance may have a constant offset from the start time of the system (global time 0) called the *phase shift*. Task instances may not "overtake" each other, i.e., instances with higher instance numbers are always released later than instances with lower instance numbers. To sum up, the attributes of an aperiodic task T (see figure 4.4) are

- minimum interarrival time iat_T
- phase shift φ_T
- release probability p_T

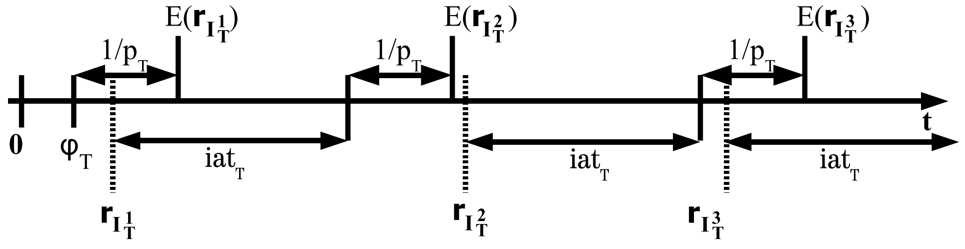


Figure 4.4: Instantiation of aperiodic task

The release time distribution is given by the conditional probability

$$Pr(r_{I_T^1} = t | r_{I_T^1} \geq t) = \begin{cases} p_T & \text{if } t \geq \varphi_T \\ 0 & \text{otherwise} \end{cases}$$

$$Pr(r_{I_T^k} = t | k > 1 \wedge r_{I_T^k} \geq t \wedge r_{I_T^{k-1}} < t) = \begin{cases} p_T & \text{if } t \geq r_{I_T^{k-1}} + iat_T \\ 0 & \text{otherwise} \end{cases}$$

Sporadic tasks have only one instance; its release time can be specified with an arbitrary distribution. For simplicity, we assume the release time of the instance of a sporadic task to be either given as an absolute number or uniformly (like a periodic task instance) or geometrically (like an aperiodic task instance) distributed.

We denote the sets of *periodic*, *aperiodic*, and *sporadic* tasks of an application by $\mathbb{T}_p$, $\mathbb{T}_a$, and T_s, respectively. Tasks of set $\mathbb{T}_a \cup \mathbb{T}_p$ release an infinite number of instances according to a predefined stochastic distribution function.

Definition 28 (Utility functions)
Each task is associated a time-discrete utility function

$$u_T : \mathbb{GT} \to \mathbb{R}_0^+.$$

For each task instance $I_T^k \in \mathbb{I}_T$, the utility function is defined as

$$u_{I_T^k} : \mathbb{GT} \to \mathbb{R}_0^+$$

with

$$u_{I_T^k}(t) = \begin{cases} 0 & \textit{if}\, t < r_{I_T^k} \\ u_T(t - r_{I_T^k}) & \textit{if}\, t \geq r_{I_T^k} \end{cases}$$

This means that $u_I(t) = 0$ for $t < r_I$, $u_I(t)$ is monotonically decreasing for $t \geq r_I$ and all instances of a task have the same shape of utility function relative to their release times, i.e.

$$\forall T \in \mathbb{T} : \forall t \in \mathbb{GT} : \forall k, k' \in \mathbb{N} : u_{I_T^k}(t + r_{I_T^k}) = u_{I_T^{k'}}(t + r_{I_T^{k'}})$$

Definition 29 (Vectors of utility functions)
For a set of task instances $\mathbb{I}'_\mathbb{T} \subseteq \mathbb{I}_\mathbb{T}$, the set of vectors of utility functions is given as

$$\mathbb{UF}_{\mathbb{I}'_\mathbb{T}} := \prod_{I \in \mathbb{I}'_\mathbb{T}} \mathbb{UF}_T \qquad \textit{with} \qquad \mathbb{UF}_T := (\mathbb{R}_0^+)^{\mathbb{GT}}$$

4.2.4 Value Functions

Value functions are defined similarly to the prior case of a single-processor system, but now on the properties of task and method instances:

Definition 30 (Value functions)
Value functions now take the following form:

$$v : \mathbb{QF}_{\mathbb{M}} \times \mathbb{UF}_{\mathbb{I}'_{\mathbb{T}}} \times \mathbb{LTF}_{\mathbb{I}'_{\mathbb{M}}} \times \mathbb{GT} \to \mathbb{R}_0^+$$

Example definitions for value functions are

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \max_{t' \le t} \left(\sum_{I_T \in \mathbb{I}'_{\mathbb{T}}} \left(u_{I_T}(t') \cdot \sum_{I_M \in c(I_T), P \in \mathbb{P}} q_M(\tau_{P,I_M}(t')) \right) \right)$$

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \sum_{I_T \in \mathbb{I}'_{\mathbb{T}}} \left(\max_{t' \le t} \left(u_{I_T}(t') \cdot \sum_{I_M \in c(I_T), P \in \mathbb{P}} q_M(\tau_{P,I_M}(t')) \right) \right)$$

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \max_{I_T \in \mathbb{I}'_{\mathbb{T}}, t' \le t} \left(u_{I_T}(t') \cdot \max_{I_M \in c(I_T), P \in \mathbb{P}} q_M(\tau_{P,I_M}(t')) \right)$$

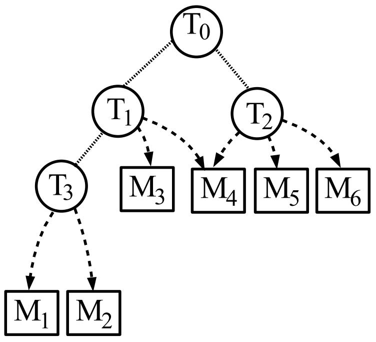
We still assume that the properties of value functions defined earlier apply accordingly.

4.3 Task Hierarchy

In the previous section, we introduced the application, library, and hardware layers for our problem description. However, the application layer still consists of an unstructured set of tasks. Now a hierarchy graph is defined on the tasks.

4.3.1 Hierarchy Graph

The hierarchy graph resembles the iterative refinement approach to software development, examples of which are the concepts of inheritance or subroutine calls. Intuitively, the hierarchy relation can express the fact that one task is a part or alternative of another. In example 4.5, the hierarchy graph contains nodes and edges on the application and library layers. Processor access specifications (edges between methods and processors) are not part of the hierarchy graph.



$T \in \mathbb{T}$	T_0	T_1
$c(T)$	$\{T_1, T_2\}$	$\{T_3, M_3, M_4\}$

$T \in \mathbb{T}$	T_2	T_3
$c(T)$	$\{M_4, M_5, M_6\}$	$\{M_1, M_2\}$

a) example graph b) child function

Figure 4.5: Example hierarchy graph

For this purpose, the child function is modified as follows:

Definition 31 (Child function for tasks in hierarchical graphs)
For task set $\mathbb{T}$ and method set $\mathbb{M}$, the child function is defined as

$$c : \mathbb{T} \to 2^{\mathbb{T} \cup \mathbb{M}}$$

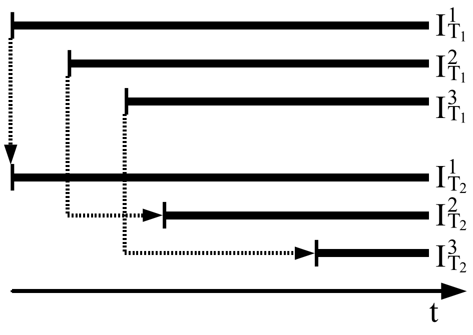
Now tasks can themselves be children of tasks; we call this a *task/subtask relationship*.

The child function defines a graph structure on the tasks and methods with the following properties:

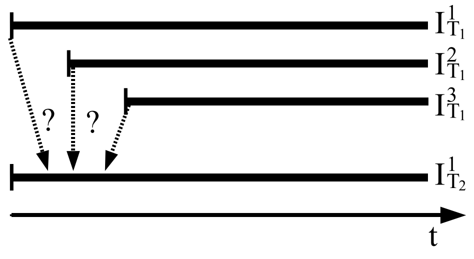
- The graph structure, restricted to task nodes, is a tree, i.e., it has a unique root node, is acyclic and provides unique parent nodes for all task nodes except for the root. Methods may be instantiated by several tasks; therefore, the tree property does not apply to the entire graph.

- The leaves of the graph are methods; all computations are ultimately based on the algorithm library, and without methods there would not be any quality functions and no notion of computational progress.

Now the question arises how to interpret the instantiation specification for tasks. For example, what should be the meaning of a periodic subtask of another periodic task (with possibly different period lengths), especially if we think of extending the parent relation to task instances? Defining a parent relationship between task instances with equal instance number would mean the release times of these instances drifting apart more and more (figure 4.6 a)); this property is not very desirable.



a) Task instances drifting apart



b) Single task instance child of multiple task instances

Figure 4.6: Instantiation problems

Having a parent task release many instances and a child task only one causes similar problems. Do we want to designate one of the parent instances to be associated with the child instance or do we introduce a parent/child relationship between all instances of the parent task and the child instance (figure 4.6b))? The first choice would leave us with no child nodes for all but one of the parent task instances, the other one would destroy the tree structure.

On the other hand, generating many instances from the child task and only one for the parent task does not cause these problems. We therefore opt to allow multiple instances of a task to be generated if and only if the same is guaranteed for all direct and indirect children and neither the parent nor the child task instances are released more frequently than the others. We can accomplish this behaviour by the following restriction on the task graph parameters and the definition of the instantiation in the following section.

Define the ancestor relation

$$a : \mathbb{T} \cup \mathbb{M} \to 2^{\mathbb{T}}$$

by

$$T \in a(N) :\Leftrightarrow T = N \vee \exists N' \in c(T) : N' \in a(N)$$

for all $N \in \mathbb{T} \cup \mathbb{M}$.

We restrict the graph structure to allow no more than one non-sporadic task on any path from the root to a leaf node. Then for all methods $M \in \mathbb{M} : |a(M) \cap (\mathbb{T}_a \cup \mathbb{T}_p)| \leq 1$.

Along with a non-sporadic task instance, the entire subtree is instantiated. A sporadic task is now not instantiated necessarily only once altogether, but *once per parent task instance*. As we will see later, the evaluation of value functions takes place in a bottom-up manner along the task hierarchy. In order to aggregate a value for the entire instance graph, it is convenient to assume the graph to be contiguous. According to the previous paragraph, this is the case if only one instance is generated from the root task, i.e., if it is sporadic. We therefore require the root task node to be sporadic; if this is in contrast to the logic of the application, a “dummy” sporadic root node can always be added.

4.3.2 Instantiation

We can then re-define the instantiation of tasks to take place as follows:

- Exactly one instance is generated from a task if the task itself and all of its ancestor nodes are sporadic.
- Infinitely many instances are generated from a task if either the task itself or one of its ancestors is non-sporadic

The set of tasks instantiated exactly once is

$$\mathbb{T}_1 := \{T \in \mathbb{T} : |a(T) \cap (\mathbb{T}_a \cup \mathbb{T}_p)| = 0\}.$$

The set of tasks instantiated infinitely often is

$$\mathbb{T}_\infty := \{T \in \mathbb{T} : |a(T) \cap (\mathbb{T}_a \cup \mathbb{T}_p)| = 1\}.$$

Then the instance set of a task T is

$$\mathbb{I}_T := \begin{cases} \{I_T^k\}_{k \in \mathbb{N}} & \text{if } T \in \mathbb{T}_\infty \\ \{I_T^1\} & \text{if } T \in \mathbb{T}_1 \end{cases}$$

During the execution of the system (the application graph together with the scheduling algorithm), an instance graph is derived from the application graph. The child function is extended to task instances:

Definition 32 (Child function for task and method instances)
For task and method instances, the child function is now defined as

$$c : \mathbb{I}_\mathbb{T} \to 2^{\mathbb{I}_\mathbb{T} \cup \mathbb{I}_\mathbb{M}}$$

$$I_T^k \in c(I_{T'}^{k'}) :\Leftrightarrow T \in c(T') \wedge (T \in (\mathbb{T}_a \cup \mathbb{T}_p) \vee k = k')$$

$$\forall M \in c(T) : \forall k' \in \mathbb{N}_0 : \exists k \in \mathbb{N}_0 : I_M^k \in c(I_T^{k'})$$

That is, together with a task, the whole subtree is instantiated. The only case for which instance numbers of parent and child task instance do not necessarily equal is that the child is an instance of a periodic or aperiodic task: The parent task by definition is instantiated only once, whereas the child task has infinitely many instances; the child instances are all children of the only parent task instance. Again, if $I_M^k \in \mathbb{I}_M$, $I_T^{k'} \in \mathbb{I}_T$ and $I_M^k \in c(I_T^{k'})$, no general statement can be made about the correlation of instance numbers. As before, method instances are generated when needed, and instance numbers are only used to tell them apart.

We do not specify release time distributions for sporadic tasks other than the precondition that release times of child task instances are no earlier than the one of the parent task instance:

$$\forall I, I' \in \mathbb{I}_{\mathbb{T}} : I' \in c(I) \Rightarrow r_{I'} \geq r_I$$

Instances of non-sporadic tasks share the same parent task instance, so that for a periodic task $T \in \mathbb{T}_p$ with parent task T' (i.e., $T \in c(T')$):

$$Pr(r_{I_T^k} = t) = \begin{cases} \frac{1}{2j_T+1} & \text{if} \quad r_{I_{T'}^1} + \varphi_T + (k-1)per_T - j_T \leq t \\ & \qquad \leq r_{I_{T'}^1} + \varphi_T + (k-1)per_T + j_T \\ 0 & \text{otherwise} \end{cases}$$

and for an aperiodic task $T \in \mathbb{T}_a$ with parent task T':

$$Pr(r_{I_T^1} = t | r_{I_T^1} \geq t) = \begin{cases} p_T & \text{if } t \geq r_{I_{T'}^1} + \varphi_T \\ 0 & \text{otherwise} \end{cases}$$

$$Pr(r_{I_T^k} = t | k > 1 \wedge r_{I_T^{k-1}} < t \wedge r_{I_T^k} \geq t) = \begin{cases} p_T & \text{if } t \geq r_{I_T^{k-1}} + iat_T \\ 0 & \text{otherwise} \end{cases}$$

4.3.3 Local Time Functions

We now introduce local time functions not only on method instances, but also on task instances. This will be necessary for scheduling algorithms to operate locally on each node, i.e., to take advantage of the task hierarchy. Using local times on the method level only requires all data on resource distribution for the entire application to be taken into account simultaneously, resulting in possibly large search spaces. Allowing the scheduler to store allocation information locally at each node of the instance graph is especially beneficial in a dynamic scheduling scheme, as it allows schedule adaptations by changing only allocations in a certain subtree at the lowest possible level without having to retouch the remainder of the schedule. The root node represents the entire application, and we assume the application can use the processors to full extent. Resource allocation takes place in top-down (i.e., the allocation to a node is distributed among the children), and evaluation in bottom-up manner, as we will see later.

First, let us define local time domains for task instances:

Definition 33 (Local time for task instances)
For task instance $I \in \mathbb{I}_{\mathbb{T}}$ and processor $P \in \mathbb{P}$, we define a local time domain as

$$\mathbb{LT}_{P,I} :\equiv \mathbb{N}_0$$

The next step is to define local time and allocation functions for task instances:

Definition 34 (Local time and allocation functions for task instances)
For all task instances $I \in \mathbb{I}_{\mathbb{T}}$, we define a local time function with regard to a certain processor as a monotonically increasing function

$$\tau_{P,I} : \mathbb{GT} \rightarrow \mathbb{LT}_{P,I} \textit{ with}$$

$$\forall t \leq r_I : \tau_{P,I}(t) = 0$$

The set of all possible local time functions for I on processor P is $\mathbb{LTF}_{P,I}$.
We define local allocation functions as

$$\alpha_{P,I} : \mathbb{LTF}_{P,I} \times \mathbb{GT} \rightarrow \mathbb{LT}_{P,I}$$

with

$$\alpha_{P,I}(\tau_{P,I}, t) := \tau_{P,I}(t+1) - \tau_{P,I}(t)$$

We define vectors of local time functions as usual:

Definition 35 (Vectors of local time functions)
For a subset $\mathbb{I}' \subseteq \mathbb{I}_{\mathbb{T}} \cup \mathbb{I}_{\mathbb{M}}$ of task and method instances, $\mathbb{LTF}_{\mathbb{I}'} := \prod_{I \in \mathbb{I}', P \in \mathbb{P}} \mathbb{LTF}_{P,I}$ is the set of vectors of local time functions for task and method instances in $\mathbb{I}'$; we use the symbol $\vec{\tau} \in \mathbb{LTF}_{\mathbb{I}'}$ for elements of this set.

The restrictions on the local time functions for methods remain valid:

- To any method instance, no more than one processor is allocated at a time.
- The sum of allocations to the set of method instances does not exceed the resources available at any time.

Apart from that, we now require that for any $\vec{\tau} \in \mathbb{LTF}_{\mathbb{I}'}$:

- The root node $I^1_{T_0}$ is allocated all processors at all times:

$$\forall P \in \mathbb{P} : \forall t \in \mathbb{GT} : \alpha_{P,I^1_{T_0}}(\tau_{P,I^1_{T_0}}, t) = 1 \qquad ^3$$

- The allocation to the children must not exceed the allocation to the parent node at any time:

$$\forall P \in \mathbb{P} : \forall t \in \mathbb{GT} : \forall I \in \mathbb{I}' : \sum_{I' \in c(I)} \alpha_{P,I'}(\tau_{P,I'}, t) \leq \alpha_{P,I}(\tau_{P,I}, t)$$

- To any task or method instance I, no processors may be allocated prior to the release time:

$$\forall I \in \mathbb{I}' : \forall P \in \mathbb{P} : \forall t \in \mathbb{GT} : t < r_I \Rightarrow \tau_{P,I}(t) = 0$$

4.3.4 And/or Graph

One advantage of hierarchisation is that the structure of the task set can now be used to impose different semantics on the parts of the graph. Instead of using one value function for the entire task set, we now construct scheduling algorithms which evaluate value functions recursively for each node in the hierarchy tree. Node value functions are of type

$$\mathbb{I}'_{\mathbb{T}} \times \mathbb{QF}_{\mathbb{M}} \times \mathbb{UF}_{\mathbb{I}'_{\mathbb{T}}} \times \mathbb{LTF}_{\mathbb{I}'_{\mathbb{T}} \cup \mathbb{I}'_{\mathbb{M}}} \times \mathbb{GT} \to \mathbb{R}^+_0$$

Specifically, we found a distinction of nodes into two logical types useful to model an important difference between them.

We assume each task (and its instances) to have one of the following types:

and: The child nodes of this task compete for the shared resources; the execution of all child node instances is desirable.

or: The child nodes of this task are alternative implementations; execution of several child node instances does in general not yield any advantage over execution of only one.

The effect of different logical types is simply the value functions that apply for each node:

[3]This definition does not take into account idle times when no method instance is actually executed on the processor. However, processing time allocated to a node in the hierarchy need not be passed on to any children (i.e., parent node allocation merely gives upper bounds to child node allocations). We opt for this notation, because it does not require prior analysis of the active set of methods at each time.

For instances I of a task with logical type *and*, we chose the value function

$$\begin{aligned} v_{I,\vec{q},\vec{u}}(\vec{\tau},t) &:= \sum_{I_T \in \mathbb{I}_{\mathbb{T}} \cap c(I)} \left(\max_{t' \le t} \left(u_I(t') \cdot v_{I_T,\vec{q},\vec{u}}(\vec{\tau},t') \right) \right) \\ &+ \sum_{I_M \in \mathbb{I}_M \cap c(I), P \in \mathbb{P}} \left(\max_{t' \le t} \left(u_I(t') \cdot q_M(\tau_{P,I_M}(t')) \right) \right) \end{aligned}$$

For example, assume we have an all-sporadic task hierarchy with the root task T_0 of logical type *and*, several child tasks $T_1, \ldots, T_n$ and associated methods $M_1, \ldots, M_n$ (figure 4.7).

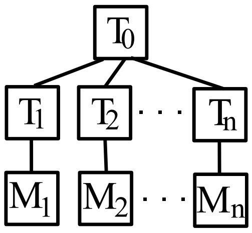


Figure 4.7: Example for *and* type task

Assume further that $r_{I^1_{T_0}} = r_{I^1_{T_1}} = \cdots = r_{I^1_{T_n}} = 0$ and $u_{I^1_{T_0}}(t) = u_{I^1_{T_1}}(t) = \cdots = u_{I^1_{T_n}}(t) = 1$ for all t. Finally, let the quality functions of all methods be concave. This setting resembles a set of anytime algorithms without timing constraints. For the child task instances T_k, we simply get

$$\begin{aligned} v_{I^1_{T_k},\vec{q},\vec{u}}(\vec{\tau},t) &= \max_{t' \le t} q_{P,M_k}(\tau_{P,I^1_{M_k}}(t')) \\ &= q_{P,M_k}(\tau_{P,I^1_{M_k}}(t)) \end{aligned}$$

The root node reflects a common objective function for the optimal stopping problem in a set of anytime tasks, where only the amount of processor time for each task (instance) is important, not the time when the allocation takes place:

$$\begin{aligned} v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau},t) &= \sum_{k=1}^{n} \max_{t' \le t} v_{I^1_{T_k},\vec{q},\vec{u}}(\vec{\tau},t') \\ &= \sum_{k=1}^{n} \max_{t' \le t} q_{P,M_k}(\tau_{P,I^1_{M_k}}(t')) \\ &= \sum_{k=1}^{n} q_{P,M_k}(\tau_{P,I^1_{M_k}}(t)) \end{aligned}$$

For instances I of a task with logical type *or*, we chose the value function

$$\begin{aligned} v_{I,\vec{q},\vec{u}}(\vec{\tau},t) \;&:=\; \max\Bigg(\max_{I_T \in \mathbb{I}_{\mathbb{T}} \cap c(I), t' \leq t} \Big(u_I(t') \cdot v_{I_T,\vec{q},\vec{u}}(\vec{\tau},t') \Big), \\ &\qquad \max_{I_M \in \mathbb{I}_M \cap c(I), P \in \mathbb{P}, t' \leq t} \Big(u_I(t') \cdot q_M(\tau_{P,I_M}(t')) \Big) \Bigg) \end{aligned}$$

Assume that the root node in the graph of figure 4.8 is of type *or*.

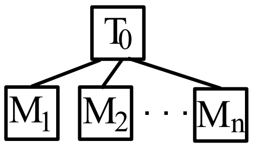


Figure 4.8: Example for *or* type task

Let $u_{T_0}(t) = 1$ for all t and define vectors of local time functions $\vec{\tau}_{(M_1)}, \dots, \vec{\tau}_{(M_n)}$ such that

$$\forall t \in \mathbb{GT} : \forall k \in \{1,\dots,n\} : \tau_{(M_k),P,I^1_{M_i}}(t) = \begin{cases} t & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

and choose $t \in \mathbb{GT}$ and $m \in \{1,\dots,n\}$ such that

$$\forall k \in \{1,\dots,n\} : v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau}_{(M_k)},t) \leq v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau}_{(M_m)},t)$$

Then for all possible vectors of local time functions $\vec{\tau}$:

$$\begin{aligned} v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau},t) &= \max_{k \in \{1,\dots,n\}, t' \leq t} q_{P,M_k}(\tau_{P,I^1_{M_k}}(t')) \\ &\leq \max_{k \in \{1,\dots,n\}, t' \leq t} q_{P,M_k}(\tau_{(M_k),P,I^1_{M_k}}(t')) \\ &= v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau}_{(M_m)},t) \end{aligned}$$

It is most rewarding to allocate all processing time to one method (instance) only; this resembles a method selection scheme.

4.3.5 Implications on Scheduling Algorithms

Scheduling algorithms now have to take into account the hierarchical structure by evaluating value functions recursively for all nodes. The overall value of the application equals the value of the root node. As an example, we re-state the pseudocode of the Simulated-Annealing algorithm with the necessary changes in figure 4.9.

```
$optimize_I(c(I), \mathbb{J}^{\geq\vartheta}_{c(I),[t_0;t_0+ws[}, J_{max}, Temp_{start}, Temp_{min}, \#rep, cF) : \vec{\tau}_{best} \in \mathbb{LTF}_{c(I)}$
  $\vec{\tau} := almostUniform(c(I), \mathbb{J}^{\geq\vartheta}_{c(I),[t_0;t_0+ws[})$
  $Temp := Temp_{start}$
  $V := v_{I,\vec{q},\vec{u}}(\vec{\tau}, J_{max})$
  $V_{best} := V$
  $\vec{\tau}_{best} := \vec{\tau}$
  while($Temp > Temp_{min}$) do
    choose one interval $J \in \mathbb{J}^{\geq\vartheta}_{c(I),[t_0;t_0+ws[}$
    for $n = 1$ to $\#rep$ do
      $\vec{\tau'} := searchStep(\vec{\tau})$
      optimise and evaluate children
      $V' := v_{I,\vec{q},\vec{u}}(\vec{\tau'}, J_{max})$
      if($V' > V$) then
        $\vec{\tau} := \vec{\tau'}$
        $V := V'$
        if $V > V_{best}$ then
          $V_{best} := V$
          $\vec{\tau}_{best} := \vec{\tau}$
        fi
      else
        with probability $\min(1, e^{\frac{V-V'}{Temp}})$ do
          $\vec{\tau} := \vec{\tau'}$
          $V := V'$
        od
      fi
      $Temp := cF \cdot Temp$
    od
  od
  return $\vec{\tau}_{best}$
end
```

Figure 4.9: Simulated-Annealing resource allocation algorithm

The optimisation algorithm is called for every node I in the task instance graph. It now distributes processor time not among all tasks, but among all child task instances $c(I)$ of the current instance I. Allocations are indexed by the processor, because the algorithm has to compute processing time distributions for a multiprocessor system. Search steps are analogous to the simpler previous case, but now must take into account the different processors. If there are multiple processors of the same kind, moving allocations from one of these processor instances to another would be possible; however, we usually forbid this kind of search steps, as they would imply task migration from one processor to another at runtime, possibly incurring enormous costs.

A central point in the new version of the algorithm is the recursive call for children of the current task. To prevent exponential costs by recalculating allocations for child nodes, we implemented caching mechanisms which store information on previously calculated partial solutions. For each node, there is a repository mapping subproblems that have been submitted to the associated optimising agent at an earlier time to the solution found. Depending on the time available for the optimisation agent when the same subproblem is re-submitted later, it may either use the previous result or try to improve on the prior solution. A metric is applied to detect similar problems to the ones previously calculated. We assume that solutions to a similar problem are similar to the solutions of the original problem, so they can be used as a starting point for optimisation. In many cases, this is a reasonable assumption; obviously, however, it is not always true.

4.3.5.1 And/or Graph

We demonstrate the implications of the and/or hierarchy of tasks with the Simulated-Annealing algorithms presented before, and we choose an example with one processor and sporadic tasks with utility functions representing firm relative deadlines δ, i.e., for every task instance I_T^k, its utility function is defined as follows:

$$u_{I_T^k}(t) = \begin{cases} 1 & \text{if } t < \delta_T \\ 0 & \text{if } t \geq \delta_T \end{cases}$$

Figure 4.10 shows an example graph with *and* type (symbol: $\wedge$) and *or* type (symbol: $\vee$) tasks. Tasks in this example are annotated with release times, r, and deadlines, δ.

Elementary intervals are calculated in bottom-up manner from the release times and utility change times (here: the deadlines) of tasks.

Figure 4.11 demonstrates the bottom-up calculation of elementary intervals for the example graph, and tables 4.1 and 4.2 the sets of elementary intervals for all instances in root-node granularity and the sets of active nodes for all elementary intervals.

Having established the set of active intervals for all tasks (figure 4.12), the optimisation algorithm starts off with a full allocation of processing time to the root node and by distributing time units in top-down manner. Figure 4.13 shows a distribution of cpu time as it might be used

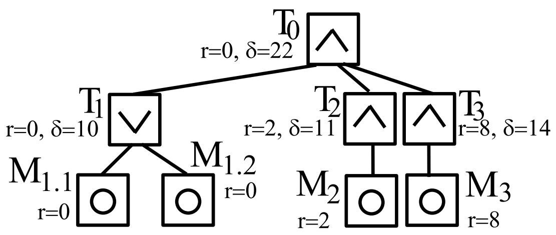


Figure 4.10: Example application graph

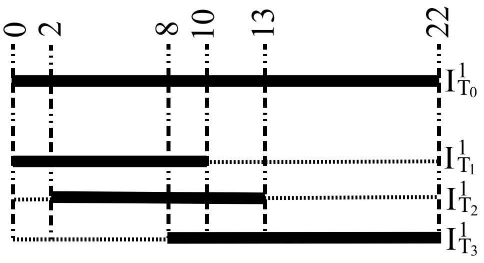


Figure 4.11: Calculation of elementary intervals for example graph

node	elementary intervals
$I^1_{T_0}$	$[0;2[, [2;8[, [8;10[, [10;13[, [13;22[$
$I^1_{T_1}/I^1_{M_{1.1}}/I^1_{M_{1.2}}$	$[0;2[, [2;8[, [8;10[$
$I^1_{T_2}/I^1_{M_2}$	$[2;8[, [8;10[, [10;13[$
$I^1_{T_3}/I^1_{M_3}$	$[8;10[, [10;13[, [13;22[$

Table 4.1: Elementary intervals for nodes of example graph

interval	active instances
$[0;2[$	$I^1_{T_0}, I^1_{T_1}, I^1_{M_{1.1}}, I^1_{M_{1.2}}$
$[2;8[$	$I^1_{T_0}, I^1_{T_1}, I^1_{M_{1.1}}, I^1_{M_{1.2}}, I^1_{T_2}, I^1_{M_2}$
$[8;10[$	$I^1_{T_0}, I^1_{T_1}, I^1_{M_{1.1}}, I^1_{M_{1.2}}, I^1_{T_2}, I^1_{M_2}, I^1_{T_3}, I^1_{M_3}$
$[10;13[$	$I^1_{T_0}, I^1_{T_2}, I^1_{M_2}, I^1_{T_3}, I^1_{M_3}$
$[13;22[$	$I^1_{T_0}, I^1_{T_3}, I^1_{M_3}$

Table 4.2: Active nodes for all elementary intervals

at the beginning of the optimisation process. To root node $I^1_{T_0}$, the allocation of cpu cycles in an interval $[t_s; t_e[$ is $t_e - t_s$ units. Allocations are distributed approximately uniformly amongst the child nodes for each task.

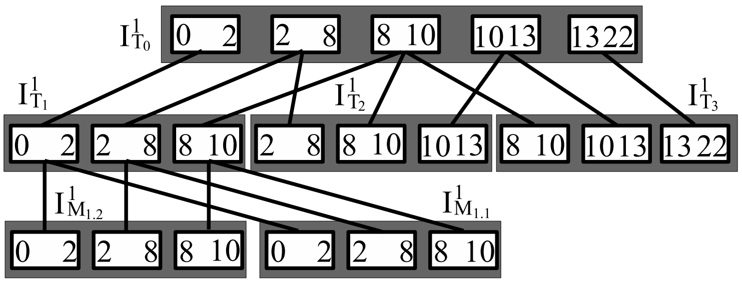


Figure 4.12: Elementary intervals for example graph

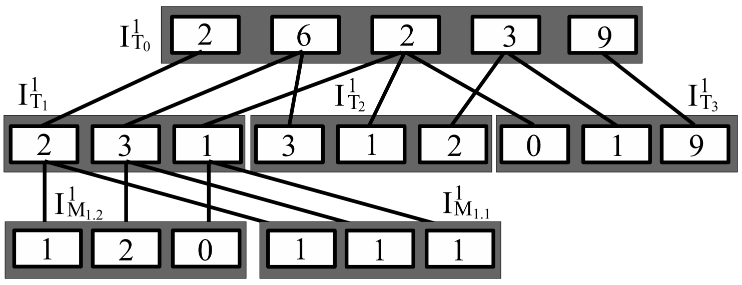


Figure 4.13: Primary distribution of cpu time

Assume the step quality functions for the method nodes as given in table 4.3, so that $q_M(t) = \max_{0 \le t' \le t} q(t')$.

Node	Step 1		Step 2		Step 3		Step 4	
	t	q	t	q	t	q	t	q
$M_{1.1}$	0	0.0	4	0.2	6	0.6		
$M_{1.2}$	0	0.0	2	0.4	8	1.0		
M_2	0	0.0	2	0.1	4	0.2	6	0.3
M_3	0	0.0	4	0.3	8	0.4	12	0.8

Table 4.3: Assumed quality functions of methods

The allocation of figure 4.13 would then yield an overall value of $\max(q_{M_{1.1}}(3), q_{M_{1.2}}(3)) + q_{M_2}(6) + q_{M_3}(10) = \max(0, 0.4) + 0.3 + 0.4 = 1.1$, the optimised allocation of figure 4.14 an overall value of $\max(q_{M_{1.1}}(0), q_{M_{1.2}}(8)) + q_{M_2}(2) + q_{M_3}(12) = \max(0, 1) + 0.1 + 0.8 = 1.9$. The Gantt chart for the resulting schedule is shown in figure 4.15.

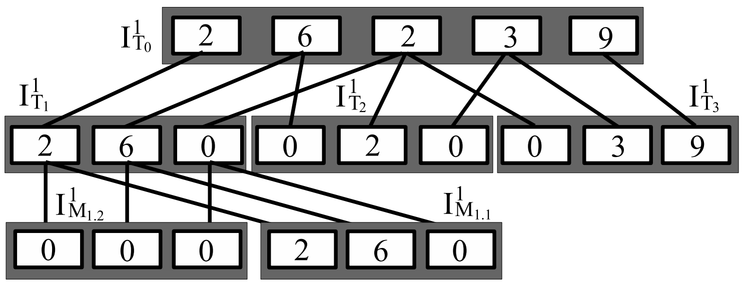


Figure 4.14: Final distribution of cpu time

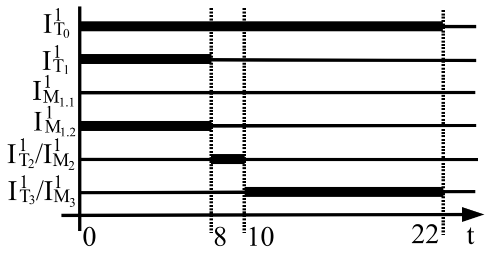


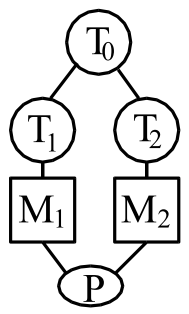
Figure 4.15: Gantt chart of result schedule

4.3.5.2 Instantiation

We demonstrate the usefulness of the distinction between tasks and instances with the MDP approach to quality/utility scheduling. The finite set of tasks with regularly occurring release times makes this scheduling algorithm practically feasible. As an example, consider the graph of figure 4.16 with root node T_0, child task nodes T_1 and T_2, methods M_1 and M_2 and one processor P.

Task T_0 is instantiated exactly once, and the release time of this instance is 0. The release times of instances of tasks T_1 and T_2 are geometrically distributed with probabilities $p_1, p_2 \in]0; 1[$ and minimum interarrival times $iat_{T_1} = iat_{T_2} = 1$. Hence, the probability for the release time of instance $I^k_{T_i}, i \in \{1, 2\}, k \in \mathbb{N}_0$ being t is

$$\begin{aligned} Pr(r_{I^1_{T_i}} = t | r_{I^1_{T_i}} \geq t) &= p_i \\ Pr(r_{I^k_{T_i}} = t | r_{I^{k-1}_{T_i}} < t \wedge r_{I^k_{T_i}} \geq t) &= p_i \end{aligned}$$



instance numbers

T_0	T_1	T_2
1	∞	∞

release probabilities

T_1	T_2
0.3	0.8

Figure 4.16: Example graph

The mean interarrival time, i.e., the expected time between two consecutive instantiations of task T_i, is

$$E(r_{I_{T_i}^{k+1}} - r_{I_{T_i}^{k}}) = \frac{1}{p_i}$$

Utility and quality functions for the example graph are given in tables 4.4, with the previously given definition for utility functions of task instances:

$$u_{I_T^k}(t) = \begin{cases} 0 & \text{if } t < r_{I_T^k} \\ u_T(t - r_{I_T^k}) & t \geq r_{I_T^k} \end{cases}$$

T_0	$\forall t \in \mathbb{GT} : u_{T_0}(t) = 1$
T_1	$u_{T_1}(t) = \begin{cases} 1 & \text{if } 0 \leq t < 2 \\ 0 & \text{if } t \geq 2 \end{cases}$
T_2	$u_{T_2}(t) = \begin{cases} 1 & \text{if } 0 \leq t < 3 \\ 0.5 & \text{if } 3 \leq t < 5 \\ 0 & \text{if } t \geq 5 \end{cases}$

a) Utility functions

M_1	$q_{P,M_1}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n \geq 1 \end{cases}$
M_2	$q_{P,M_2}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 0.1 & \text{if } n = 1 \\ 0.5 & \text{if } n = 2 \\ 1 & \text{if } n \geq 3 \end{cases}$

b) Quality functions

Table 4.4: Quality and utility functions for example

For the MDP formulation of the problem, let states be denoted by a tuple $(t, \langle a_1, \ldots, a_t \rangle, \mathbb{I}(t))$, where $\mathbb{I}(t)$ is the set of ready instances at time t. At any time t, at most one new instance of tasks T_1 and T_2 can arrive. The probabilities are $0.7 \cdot 0.2 = 0.14$ for no new instance being released, $0.3 \cdot 0.2 = 0.06$ for only a new instance of T_1, $0.7 \cdot 0.8 = 0.56$ for only a new instance of T_2 being released. The probability of instances of either task arriving is

$0.3 \cdot 0.8 = 0.24$. Note that the partial schedules do not have any influence on the transition probabilities between states, so that forward simulation to find the states most likely to be encountered can be done on state sets with variables being used within the partial schedules, as demonstrated in figure 4.17. The state sets $S_1, \ldots, S_9$ are numbered in the order they are encountered during forward simulation. The forward simulation algorithm always expands the state transition graph with transitions from the state set with highest probability (the product of transition probabilities on the path originating from state s_0). Edges are annotated with transition probabilities and the set of tasks that are instantiated during this transition; remember that at most one instance of each task is released at each time step. Nodes are annotated with the probability of entering a state (set) from start state s_0 and the name of the state set; only the most likely state sets are assigned names.

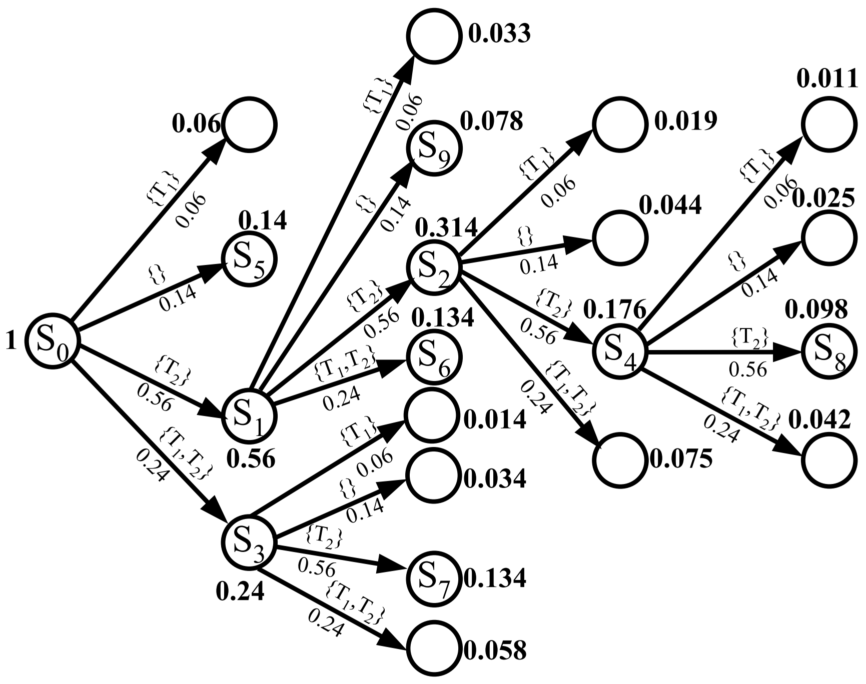


Figure 4.17: Forward simulation

Table 4.5 lists the state sets with their associated attributes, i.e., time, set of ready task instances and partial schedule. The first column contains the name of a state or the structural description for a set of states (using variables instead of individual actions within the partial schedules), the probability of reaching this state (set) starting from state s_0 and the current action. The possible transitions from each state set are given in column 2, followed by the probability of reaching the direct successor state (set) starting from state s_0. The most likely successor state sets are marked by an asterisk ($*$) and named (last column). The notation $I_{T_i}^{k(t)}$ is used to indicate the k-th instance of T_i arriving at time t. An entry ϵ in a schedule means leaving the processor idle.

State (pattern)		Transitions	Successor	Successor
prob.	action		state prob.	state set
s_0		$\xrightarrow{0.14} (1, \langle\rangle, \emptyset)$	$0.14*$	S_5
		$\xrightarrow{0.06} (1, \langle\rangle, \{I_{T_1}^{1(1)}\})$	0.06	
		$\xrightarrow{0.56} (1, \langle\rangle, \{I_{T_2}^{1(1)}\})$	$0.56*$	S_1
1	ϵ	$\xrightarrow{0.24} (1, \langle\rangle, \{I_{T_1}^{1(1)}, I_{T_2}^{1(1)}\})$	$0.24*$	S_3
$(1, \langle\rangle,$ $\{I_{T_2}^{1(1)}\})$		$\xrightarrow{0.14} (2, \langle x_1\rangle, \{I_{T_2}^{1(1)}\})$	$0.0784*$	S_9
		$\xrightarrow{0.06} (2, \langle x_1\rangle, \{I_{T_1}^{1(2)}, I_{T_2}^{1(1)}\})$	0.0336	
		$\xrightarrow{0.56} (2, \langle x_1\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$	$0.3136*$	S_2
0.56	x_1	$\xrightarrow{0.24} (2, \langle x_1\rangle, \{I_{T_1}^{1(2)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$	$0.1344*$	S_6
$(2, \langle x_1\rangle,$ $\{I_{T_2}^{1(1)},$ $I_{T_2}^{2(2)}\})$		$\xrightarrow{0.14} (3, \langle x_1x_2\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$	0.0439	
		$\xrightarrow{0.06} (3, \langle x_1x_2\rangle, \{I_{T_1}^{1(3)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$	0.0188	
		$\xrightarrow{0.56} (3, \langle x_1x_2\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\})$	$0.1756*$	S_4
0.31	x_2	$\xrightarrow{0.24} (3, \langle x_1x_2\rangle, \{I_{T_1}^{1(3)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\})$	0.0753	
$(1, \langle\rangle,$ $\{I_{T_1}^{1(1)},$ $I_{T_2}^{1(1)}\})$		$\xrightarrow{0.14} (2, \langle x_1'\rangle, \{I_{T_1}^{1(1)}, I_{T_2}^{1(1)}\})$	0.0336	
		$\xrightarrow{0.06} (2, \langle x_1'\rangle, \{I_{T_1}^{1(1)}, I_{T_1}^{2(2)}, I_{T_2}^{1(1)}\})$	0.0144	
		$\xrightarrow{0.56} (2, \langle x_1'\rangle, \{I_{T_1}^{1(1)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$	$0.1344*$	S_7
0.24	x_1'	$\xrightarrow{0.24} (2, \langle x_1'\rangle, \{I_{T_1}^{1(1)}, I_{T_1}^{2(2)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$	0.0576	
$(3, \langle x_1x_2\rangle,$ $\{I_{T_2}^{1(1)},$ $I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\}$		$\xrightarrow{0.14} (4, \langle x_1x_2x_3\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\})$	0.0246	
		$\xrightarrow{0.06} (4, \langle x_1x_2x_3\rangle, \{I_{T_1}^{1(4)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\})$	0.0105	
		$\xrightarrow{0.56} (4, \langle x_1x_2x_3\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$	$0.0983*$	S_8
0.18	x_3	$\xrightarrow{0.24} (4, \langle x_1x_2x_3\rangle, \{I_{T_1}^{1(4)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$	0.0421	
$x_1 \in \left\{\epsilon, I_{T_2}^{1(1)}\right\}, x_1' \in \left\{\epsilon, I_{T_1}^{1(1)}, I_{T_2}^{1(1)}\right\}$ $x_2 \in \left\{\epsilon, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\right\}, x_3 \in \left\{\epsilon, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\right\}$				

Table 4.5: Forward simulation of state sets

We expand the most likely encountered state sets into the individual states of the state envelope by replacing the variables in the structural description of the state sets with their possible values (table 4.6). We avoid to bind any variables to ϵ if other values are possible. As our assumption is that all tasks are preemptive and we neglect context switch costs, leaving the processor idle in the presence of ready task instances could not yield any advantage. Note that we add the absorbing states s_{out} and s_{err}. States s_0, s_{out} and s_{err} do not have an internal structure.

State set	State	Structure
$--$	s_0	$--$
S_5	s_1	$(1, \langle\rangle, \emptyset)$
S_1	s_2	$(1, \langle\rangle, \{I_{T_2}^{1(1)}\})$
S_3	s_3	$(1, \langle\rangle, \{I_{T_1}^{1(1)}, I_{T_2}^{1(1)}\})$
S_9	s_4	$(2, \langle I_{T_2}^{1(1)}\rangle, \{I_{T_2}^{1(1)}\})$
S_2	s_5	$(2, \langle I_{T_2}^{1(1)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$
S_6	s_6	$(2, \langle I_{T_2}^{1(1)}\rangle, \{I_{T_1}^{1(2)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$
S_7	s_7	$(2, \langle I_{T_1}^{1(1)}\rangle, \{I_{T_1}^{1(1)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$
S_7	s_8	$(2, \langle I_{T_2}^{1(1)}\rangle, \{I_{T_1}^{1(1)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\})$
S_4	s_9	$(3, \langle I_{T_2}^{1(1)}, I_{T_2}^{1(1)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\})$
S_4	s_{10}	$(3, \langle I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\})$
S_8	s_{11}	$(4, \langle I_{T_2}^{1(1)}, I_{T_2}^{1(1)}, I_{T_2}^{1(1)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$
S_8	s_{12}	$(4, \langle I_{T_2}^{1(1)}, I_{T_2}^{1(1)}, I_{T_2}^{2(2)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$
S_8	s_{13}	$(4, \langle I_{T_2}^{1(1)}, I_{T_2}^{1(1)}, I_{T_2}^{3(3)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$
S_8	s_{14}	$(4, \langle I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{1(1)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$
S_8	s_{15}	$(4, \langle I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{2(2)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$
S_8	s_{16}	$(4, \langle I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}\rangle, \{I_{T_2}^{1(1)}, I_{T_2}^{2(2)}, I_{T_2}^{3(3)}, I_{T_2}^{4(4)}\})$
$--$	s_{out}	$--$
$--$	s_{err}	$--$

Table 4.6: Names and structure of states

After that, we calculate the reward of every state by evaluating the appropriate utility and quality functions for the given partial schedule (table 4.7); note that in this example, state s_0 is associated with (global) time 0, so the first task instances may appear at time 1. The partial schedules contained in the attributes of the states refer to the situation exactly one unit of time prior to the time indicated by the time attribute of the state. This is to say that we allow task instances to be scheduled only at the time instant immediately after their release.[4] Utility and

[4]The alternative would be to allow all instances possibly released at a time instant to be included in the schedule immediately, making the search space bigger and the state transition table more complex.

quality functions are evaluated according to the definition of the value function in section 2.5.2 (sum of product of quality and utility with inner hold operator). All states except s_0, s_{out} and s_{err} have a unique predecessor state, such that the reward can be defined as the difference between a state's value and its predecessor's value.

State(s)	v_{I_0}	R
s_0	$-$	**0**
s_1, s_2, s_3	**0**	**0**
s_4, s_5, s_6, s_8	$q_{P,M_2}(1)u_{T_2}(1) = 0.1 \cdot 1 = \mathbf{0.1}$	$0.1 - 0 = \mathbf{0.1}$
s_7	$q_{P,M_1}(1)u_{T_1}(1) = 1 \cdot 1 = \mathbf{1}$	$1 - 0 = \mathbf{1.0}$
s_9	$q_{P,M_2}(2)u_{T_2}(2) = 0.5 \cdot 1 = \mathbf{0.5}$	$0.5 - 0.1 = \mathbf{0.4}$
s_{10}	$q_{P,M_2}(1)u_{T_2}(1) + q_{P,M_2}(1)u_{T_2}(1)$ $= 0.1 \cdot 1 + 0.1 \cdot 1 = \mathbf{0.2}$	$0.2 - 0.1 = \mathbf{0.1}$
s_{11}	$q_{P,M_2}(3)u_{T_2}(3) = 1 \cdot 0.5 = \mathbf{0.5}$	$0.5 - 0.5 = \mathbf{0}$
s_{12}	$q_{P,M_2}(2)u_{T_2}(2) + q_{P,M_2}(1)u_{T_2}(2)$ $= 0.5 \cdot 1 + 0.1 \cdot 1 = \mathbf{0.6}$	$0.6 - 0.5 = \mathbf{0.1}$
s_{13}	$q_{P,M_2}(2)u_{T_2}(2) + q_{P,M_2}(1)u_{T_2}(1)$ $= 0.5 \cdot 1 + 0.1 \cdot 1 = \mathbf{0.6}$	$0.6 - 0.5 = \mathbf{0.1}$
s_{14}	$q_{P,M_2}(2)u_{T_2}(3) + q_{P,M_2}(1)u_{T_2}(1)$ $0.5 \cdot 0.5 + 0.1 \cdot 1 = \mathbf{0.35}$	$0.35 - 0.2 = \mathbf{0.15}$
s_{15}	$q_{P,M_2}(1)u_{T_2}(1) + q_{P,M_2}(2)u_{T_2}(2)$ $0.1 \cdot 1 + 0.5 \cdot 1 = \mathbf{0.6}$	$0.6 - 0.2 = \mathbf{0.4}$
s_{16}	$q_{P,M_2}(1)u_{T_2}(1) + q_{P,M_2}(1)u_{T_2}(1)$ $+q_{P,M_2}(1)u_{T_2}(1) = 3 \cdot 0.1 \cdot 1 = \mathbf{0.3}$	$0.3 - 0.2 = \mathbf{0.1}$
s_{out}	$-$	**0**
s_{err}	$-$	$-\infty$

Table 4.7: Rewards for states in state envelope

We can then derive the state transition table 4.8 from figure 4.17 and table 4.6.

The only cycles in the transition graph are the self-loops of the absorbing states. Remember that

$$s \xrightarrow{(a,p)} s_{out}$$

if a is a legal action for state s and $p = 1 - \sum_{s' \in \hat{\mathbb{S}} : s \xrightarrow{(a,p')} s'} p'$ and for all states $s' \in \hat{\mathbb{S}} : time_{s'} = time_s + 1$ [5],

[5] $time_s$ is the time attribute in state s

state	possible transitions
s_0	$s_0 \xrightarrow{(\epsilon,0.14)} s_1, s_0 \xrightarrow{(\epsilon,0.56)} s_2, s_0 \xrightarrow{(\epsilon,0.24)} s_3$
s_2	$s_2 \xrightarrow{(I_{T_2}^{1(1)},0.14)} s_4, s_2 \xrightarrow{(I_{T_2}^{1(1)},0.56)} s_5, s_2 \xrightarrow{(I_{T_2}^{1(1)},0.24)} s_6$
s_3	$s_3 \xrightarrow{(I_{T_1}^{1(1)},0.56)} s_7, s_3 \xrightarrow{(I_{T_2}^{1(1)},0.56)} s_8$
s_5	$s_5 \xrightarrow{(I_{T_2}^{1(1)},0.56)} s_9, s_5 \xrightarrow{(I_{T_2}^{2(2)},0.56)} s_{10}$
s_9	$s_9 \xrightarrow{(I_{T_2}^{1(1)},0.56)} s_{11}, s_9 \xrightarrow{(I_{T_2}^{2(2)},0.56)} s_{12}, s_9 \xrightarrow{(I_{T_2}^{3(3)},0.56)} s_{13}$
s_{10}	$s_{10} \xrightarrow{(I_{T_2}^{1(1)},0.56)} s_{14}, s_{10} \xrightarrow{(I_{T_2}^{2(2)},0.56)} s_{15}, s_{10} \xrightarrow{(I_{T_2}^{3(3)},0.56)} s_{16}$

Table 4.8: State transition table

$$s \xrightarrow{(a,1)} s_{err}$$

if a is not a legal action in state s (i.e., it is not contained in the set of ready task instances) and $time_{s'} = time_s + 1$ and

$$s \xrightarrow{(a,0)} s'$$

if s' is not a legal successor state of s with regard to the time attributes or the partial schedules.

For the policy iteration, we choose a discount factor of 0.9, start with a policy $\pi(s) = \epsilon$ for all states s and have to solve the following linear equation system:

$V_\pi(s_0) = 0 + 0.9 \cdot (0.14V_\pi(s_1) + 0.56V_\pi(s_2) + 0.24V_\pi(s_3))$
$\forall s \in \{s_1, \ldots, s_{16}, s_{out}, s_{err}\} : V_\pi(s) = R(s)$

Obviously, the trivial solution to this system is

$$\forall s \in \{s_0, \ldots, s_{16}, s_{out}, s_{err}\} : V_\pi(s) = R(s).$$

Iterating through all states of the envelope, we find that, e.g., $I_{T_1}^{1(1)}$ would be a better action for state s_3 than ϵ, as

$$\begin{aligned}(R(s_3) + \gamma \sum_{s' \in \hat{\mathbb{S}}} Pr(s_3, I_{T_1}^{1(1)}, s')V_\pi(s'))\\ = 0 + 0.9 \cdot 0.56 \cdot 1 = 0.504 > 0 = V_\pi(s_3)\end{aligned}$$

The policy we receive after a sufficient series of three runs of the policy iteration algorithm (with the additional heuristic that executing any action in states on the border of the envelope is better than the null action ϵ) is as follows:

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	s_{out}	s_{err}	
ϵ	ϵ	$I_{T_2}^{1(1)}$	$I_{T_1}^{1(1)}$	$I_{T_2}^{1(1)}$	$I_{T_2}^{1(1)}$	$I_{T_1}^{1(1)}$	$I_{T_2}^{2(2)}$	$I_{T_1}^{1(1)}$	$I_{T_2}^{3(3)}$	$I_{T_2}^{2(2)}$	$I_{T_2}^{2(2)}$	$I_{T_2}^{2(2)}$	$I_{T_2}^{3(3)}$	$I_{T_2}^{2(2)}$	$I_{T_2}^{2(2)}$	$I_{T_2}^{3(3)}$	ϵ	ϵ	

This policy guides the actions of the scheduler; a new policy has to be calculated once the current state is not contained in the envelope any more. Figure 4.17 demonstrates that the state envelope and the corresponding policy is sufficient for the next time step with a probability of $0.14 + 0.56 + 0.24 = 0.94$, for the following time step with a probability of $0.078 + 0.314 + 0.134 + 0.134 = 0.66$, and for the time step of distance 3 to the current time with probability 0.176.

4.3.6 Caching Mechanism

The hierarchisation of the application graph and the adaptations of the scheduling algorithms we introduced earlier would require subtrees of the graph to be evaluated recursively, even if the local problems in some subgraphs remain unchanged. To avoid this drawback, we use local caches at each node to store problem descriptions encountered before and use the previously calculated solutions, thus pruning the recursion tree if possible. Along with this basic scheme, we also introduced a further improvement of scheduling algorithms storing solutions not only on the level of task/method instance graphs, but also on the level of the original problem description (the task/method graph). This enables the scheduler to use solutions of prior instantiations of tasks and entire subgraphs; this is only appropriate, however, if the relative release times of tasks within the subgraph are well predictable. Finally, we can use the cache not only to detect subproblems which have previously been solved, but also to find problems similar to the current one, hoping to receive a better starting point for the search algorithm than an entirely arbitrary resource distribution.

4.4 Dependencies

Dependency edges act as a value-based equivalent of dataflow or other precedence constraints.

4.4.1 Dependencies on the Task / Method Level

The dependency graph is a second directed graph structure defined on the same task nodes as the hierarchy graph and is expressed by a predecessor function. We do not allow dependency edges between methods, as this would prevent reusability with the same arguments we used to justify the restriction of no method instance being the child of two different task instances. All desired dependencies must be expressed on the task level; this may in some cases require the introduction of dummy tasks.

$$pred : \mathbb{T} \rightarrow 2^{\mathbb{T}}$$

Hierarchy and dependency graphs are in some regard orthogonal to each other, as dependency edges cannot connect tasks on different levels of the hierarchy tree.

To allow evaluation to still take place recursively along the task hierarchy and due to the semantics of different kinds of task nodes, certain restrictions apply to the dependency graph:

- Dependencies are allowed only between child nodes of the same *and* type node.

  The restriction to *and* type nodes was introduced primarily for semantic reasons; the interpretation of an *or* type node is such that the child nodes represent alternative implementations of the parent. It does not seem to make sense in most cases that one alternative depends on another one; in general, we want only one of the alternatives to be executed at all.

  The restriction of dependencies to nodes sharing the same parent is necessary to keep value function evaluation efficient; allowing dependencies to span several hierarchy levels would mean we cannot aggregate values hierarchically any more with significant consequences for the efficiency of scheduling algorithms.

- Dependencies are only allowed between tasks which are not both non-sporadic.

  Allowing dependencies between non-sporadic tasks would lead us to similar difficulties as we described for the task hierarchy. For example, a dependency between periodic tasks of different period lengths would involve correlated instances diverging in time more and more, which we intend to avoid. A sporadic task may, on the other hand, trigger the start of repetitive computations, and a certain minimum number of executed instances of a non-sporadic task may trigger a one-time action.

  The dependency graph on the task level acts as an abbreviation for the equivalent relationship on the instance level. A distinction must be made, however, according to the instantiation type of tasks. If either the source or the target node of a dependency edge is non-sporadic, an infinite number of instances are created from one task, but only one instance from the other task. Each instance of these tasks takes part in the same dependency relationship, regardless of the instance numbers. This implies that dependency cycles involving a non-sporadic task would mean each instance of this task to depend on all instances of the same task, including itself. This would question causality, so that we have to forbid cycles including non-sporadic tasks.

For sporadic tasks, however, the situation is different: A cycle in the task-level dependency graph means that an instance of a task depends on another instance of the same task. We suppose this makes sense as long as the dependency refers to an earlier instance. This leads us to the following statement:

- A delay specification for sporadic tasks is defined on the set of dependency edges as follows:

  $$delay : \mathbb{T}_s \times \mathbb{T}_s \to \mathbb{N}_0$$

  with

  $$delay(T, T') \begin{cases} = \infty & \text{if } T \notin pred(T') \\ \in \mathbb{N}_0 & \text{if } T \in pred(T') \end{cases}$$

  The delay is the distance in instance numbers for task instances depending on each other. For example, if $delay(T, T') = k$, the $(i + k)$-th instance of task T' depends on the $i-$th instance of task T. For $i \leq k$, $I^i_{T'}$ does not depend on any instance of task T.

  For every cycle in the dependency graph, the delay specification is positive: Assume there are $T_1, \ldots, T_n \in \mathbb{T}$ with $T_n \in pred(T_1)$ and $\forall j \in \{2, \ldots, n\} : T_{j-1} \in pred(T_j)$.
  Then

  $$delay(T_n, T_1) + \sum_{j=1}^{n} delay(T_{j-1}, T_j) > 0$$

A positive weight $weight(T, T')$ is assigned to nodes connected by a dependency edge, i.e.,

$$weight : \mathbb{T} \times \mathbb{T} \to [0; 1]$$

with

$$weight(T, T') \begin{cases} > 0 & \text{if } T \in pred(T') \\ = 0 & \text{otherwise} \end{cases}$$

An example application graph with task hierarchy, dependency graph, periodicity specification and a multiprocessor target architecture is shown in figure 4.18.

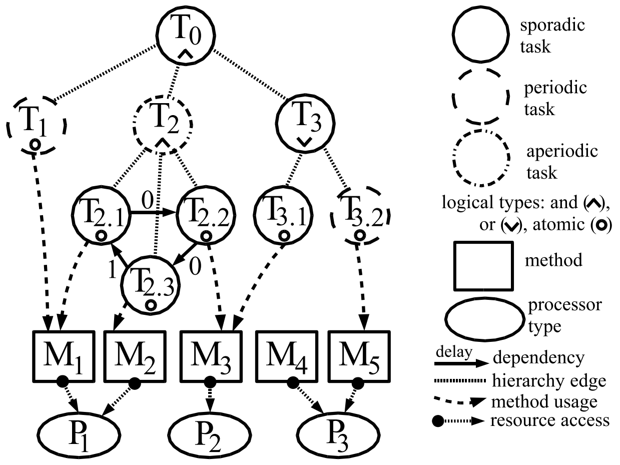


Figure 4.18: Example application graph

4.4.2 Dependencies on the Instance Level

Similarly to the hierarchy graph, a dependency graph is constructed between the task instances. Dependencies between instances have to take into account the delay specifications:

$$pred : \mathbb{I}_{\mathbb{T}} \rightarrow 2^{\mathbb{I}_{\mathbb{T}}}$$

For dependency edges between task instances, the following must be true for $I_T^k, I_{T'}^{k'} \in \mathbb{I}_{\mathbb{T}}$:

$$I_{T'}^{k'} \in pred(I_T^k) :\Leftrightarrow T' \in pred(T) \wedge k - k' = delay(T', T)$$

Finally, we define weights for the dependency edges indicating the level of influence of the predecessor on the successor node:

$$I_{T'}^{i'} \in pred(I_T^i) \Rightarrow weight(I_{T'}^{i'}, I_T^i) := weight(T', T)$$

The delay specification is necessary in order to guarantee the instance dependency graph to be acyclic, as we will show now:

Assume $pred : \mathbb{I}_{\mathbb{T}_s} \rightarrow 2^{\mathbb{I}_{\mathbb{T}_s}}$ to define a cyclic instance dependency graph. Then the following must be true:

$$\exists T_1, \ldots, T_n \in \mathbb{T} : \exists k_1, \ldots, k_n : T_n^{k_n} \in pred(T_1^{k_1}) \wedge \forall_{i=2}^n T_{i-1}^{k_{i-1}} \in pred(T_i^{k_i})$$

From the definition of dependency edges on instances, we know that

$$T_n \in pred(T_1) \wedge \forall_{i=2}^n : T_{i-1} \in pred(T_i)$$

and

$$k_1 - k_n = delay(T_n, T_1) \wedge \forall_{i=2}^{n} : k_i - k_{i-1} = delay(T_{i-1}, T_i)$$

We receive

$$\begin{aligned} delay(T_n, T_1) + \sum_{i=2}^{n} delay(T_{i-1}, T_i) &= k_1 - k_n + \sum_{i=2}^{n} (k_i - k_{i-1}) \\ &= (k_1 + \sum_{i=2}^{n} k_i) - (k_n + \sum_{i=2}^{n} k_{i-1}) \\ &= \sum_{i=1}^{n} k_i - \sum_{i=1}^{n} k_i = 0, \end{aligned}$$

in contradiction to the requirements for the delay specification.

□

For example, the application graph of figure 4.18 unfolds partially into the instance graph of figure 4.19. Only up to three of a possibly infinite number of instances of the tasks are shown. The dynamic scheduler repeatedly constructs partial instance graphs according to its limited view of the future.

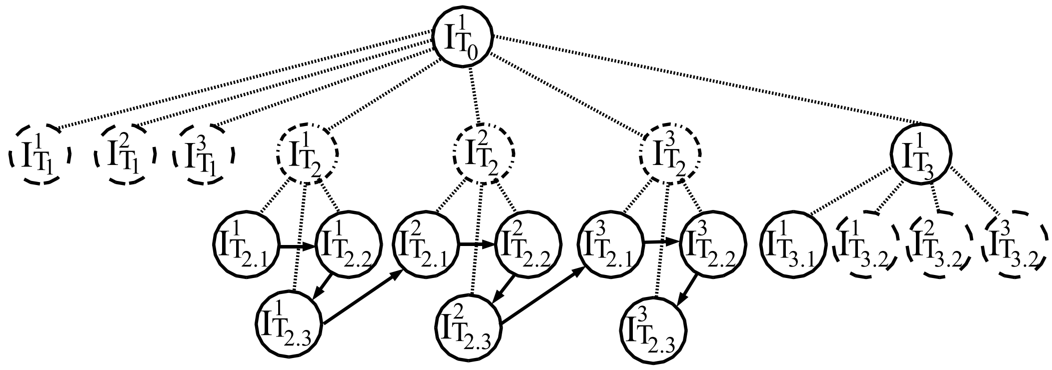


Figure 4.19: Partial instance graph

Note that in the context of value-based scheduling, there is no intrinsic equivalent to the completion of execution in traditional task models. We can, however, emulate methods with fixed execution time n_0 in our model by using quality functions with two values only, e.g.,

$$q(n) = \begin{cases} 0 & \text{if } n < n_0 \\ 1 & \text{if } n \geq n_0 \end{cases}$$

In this regard, the situation described here, where quality functions are not restricted to this shape, is more general than precedence conditions.

Our model assumes that all tasks can be executed independently of each other. Dependencies expressed through weighted edges between the task nodes affect only the evaluation of value functions and hence the overall value achieved by the application. We call this kind of relationship *value dependency*.

Imagine task instance I_1 has to be executed prior to I_2 because of a precedence constraint. Our schedulers would then assign a smaller value to the pair of tasks when executed the other way around; hence, the optimisation algorithm would effectively avoid this situation.

4.4.3 Value Functions

We want to introduce an interpretation of dependencies which allows us to map precedence constraints into our system of value functions without having to resort to constraint solving techniques for this aspect of the scheduling problem. The basic idea is to modify the definition of value functions such that they penalise schedules which include the execution of tasks in an order contradicting the desired precedence. We then rely on the value-based scheduler to find schedules with high values. Depending on the performance of the scheduling algorithm, we can hope that most precedence constraints are fulfilled. However, we cannot guarantee that all constraints are met. Modelling dataflow constraints with value functions, our schedulers would allow to execute tasks in the wrong order; however, this would result in the system or at least the subsystem in question not being able to achieve any positive value.

For the moment, assume all edge weights to equal 1. We define the new value function as

$$v^*_{I,\vec{q},\vec{u}}(\vec{\tau},t) := \left(\max_{t' \leq t} v_{I,\vec{q},\vec{u}}(\vec{\tau},t') \cdot \prod_{I' \in pred(I)} v^*_{I',\vec{q},\vec{u}}(\vec{\tau},t') \right) \quad [6]$$

Note that this recursive definition only terminates for acyclic instance dependency graphs, which is guaranteed by the delay constraints.

Let us now look at several typical patterns which frequently occur in a dependency graph.

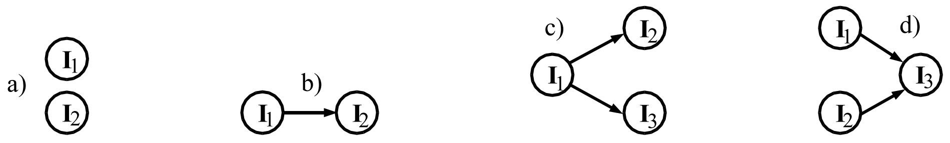


Figure 4.20: Typical patterns

[6]with $\prod_{\emptyset} expr = 1$ for an arbitrary expression $expr$

Independent instances: For two independent task instances (figure 4.20a)), I_1 and I_2, the value functions are the original ones:

$$v^*_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t)$$

$$v^*_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t)$$

Sequence of instances: If two instances form a sequence (figure 4.20b)), the value of I_1 is independent of the value of I_2, but the value of I_2 is influenced by the value of I_1.

$$v^*_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t)$$

$$v^*_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t) = \max_{t' \leq t} \Big(v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t') \cdot v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t') \Big)$$

A value of 0 for I_1 (i.e., I_1 has not made any recognisable progress) means that the successor I_2 cannot achieve any value:

$$v^*_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t) = \max_{t' \leq t}(0 \cdot v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t)) = 0$$

Fork: If three instances form a fork pattern (figure 4.20c)), the value of I_1 is independent of the other instances and the values of I_2 and I_3 are influenced equally by the value of I_1.

$$v^*_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t)$$

$$v^*_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t) = \max_{t' \leq t} \Big(v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t') \cdot v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t') \Big)$$

$$v^*_{I_3,\vec{q},\vec{u}}(\vec{\tau}, t) = \max_{t' \leq t} \Big(v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t') \cdot v_{I_3,\vec{q},\vec{u}}(\vec{\tau}, t') \Big)$$

Join: Finally, if three instances form a join pattern (figure 4.20c)), the values of I_1 and I_2 are independent of the value of I_3, and I_3 is influenced by the values of both I_1 and I_2.

$$v^*_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t)$$

$$v^*_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t)$$

$$v^*_{I_3,\vec{q},\vec{u}}(\vec{\tau}, t) = \max_{t' \leq t} \Big(v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t') \cdot v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t') \cdot v_{I_3,\vec{q},\vec{u}}(\vec{\tau}, t') \Big)$$

Maximum values at the source nodes mean that the successor node can also achieve maximum performance. If only one of the predecessors has 0 value, no value can be expected from the successor node.

Traditional dataflow between run-to-completion tasks means that the successor task can only start once the predecessors have commenced execution. Suppose we model run-to-completion tasks as before by two-valued quality functions. In our model, the successor node can always start execution; it cannot, however, achieve any positive value if the successors have not finished. Triggering behaviour, where only one of the predecessors is needed as an input, can be implemented via the task hierarchy with *or* type tasks.

Note that above definition of v^* does not yet consider edge weights $\neq 1$. In order to include edge weights into the calculation of aggregate functions, we first pose a series of conditions on an appropriately extended definition.

4.4.4 Interpreting Edge Weights

The edge weight expresses a level of influence the predecessor has on the successor node. As with the original definition of value functions, we now state a series of properties which must be true for value functions $v^\ddagger$ within weighted directed dependency graphs.

- For source nodes of the instance dependency graph, the new function equals the original value function, i.e.
$$pred(I) = \emptyset \Rightarrow v^\ddagger_{I,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I,\vec{q},\vec{u}}(\vec{\tau}, t)$$
- A higher edge weight must not result in a smaller impact of the predecessor node on the successor node. Consider two task instances, I_1 and I_2, with only one predecessor node, i.e., $pred(I_1) = pred(I_2) = \{I\}$. Assume $weight(I, I_1) \leq weight(I, I_2)$. Assume further that $v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t) = v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t)$ and $\tau_{P,I_1} = \tau_{P,I_2}$ for all $P \in \mathbb{P}$.

  Then the following must be true:
$$\forall t \in \mathbb{GT} : \forall \vec{\tau} \in \mathbb{LTF}_{\mathbb{I_T} \cup \mathbb{I_M}} : |v^\ddagger_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t) - v_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t)| \leq |v^\ddagger_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t) - v_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t)|$$
- A higher value at the source node of an edge must not result in a lower value at the target node of the edge. Consider two task instances, I_1 and I_2. Assume there are task instances $I'_1 \in pred(I_1)$ and $I'_2 \in pred(I_2)$, such that $I'_1 \neq I'_2$ and $pred(I_1)\backslash\{I'_1\} = pred(I_2)\backslash\{I'_2\}$. Assume further that $weight(I'_1, I_1) = weight(I'_2, I_2)$ and for all common predecessor nodes $I \in pred(I_1) \cap pred(I_2)$: $weight(I, I_1) = weight(I, I_2)$. Then the following must be true:
$$\forall t \in \mathbb{GT} : \forall \vec{\tau} \in \mathbb{LTF}_{\mathbb{I_T} \cup \mathbb{I_M}} : v^\ddagger_{I'_1,\vec{q},\vec{u}}(\vec{\tau}, t) \leq v^\ddagger_{I'_2,\vec{q},\vec{u}}(\vec{\tau}, t) \Leftrightarrow v^\ddagger_{I_1,\vec{q},\vec{u}}(\vec{\tau}, t) \leq v^\ddagger_{I_2,\vec{q},\vec{u}}(\vec{\tau}, t)$$

We chose the following definition for the value function $v^{\ddagger}$, which holds all of the properties listed above:

$$v^{\ddagger}_{I,\vec{q},\vec{u}}(\vec{\tau},t) := \max_{t' \leq t} \left(v_{I,\vec{q},\vec{u}}(\vec{\tau},t') \cdot \prod_{I' \in pred(I)} v^{\ddagger}_{I',\vec{q},\vec{u}}(\vec{\tau},t')^{weight(I',I)} \right)$$

4.4.5 Limiting the Scope of Value Dependencies

A useful simplification for the calculation of value functions in the presence of value dependencies is to limit the scope of the dependencies. We assume that changing values at the source node of an edge do not necessarily always affect the target node of the edge, but that a "value flow" takes place only once, namely at the time of the first allocation of resources to the target task instance. This is a further analogy to dataflow modelling, where it is usually assumed that the data are completely read before the actor starts its own calculations. In this case, the influence of the predecessor nodes can be represented by a scalar, the impact factor $\xi_{I,\vec{\tau}} \in \mathbb{R}_0^+$, defined as:

$$\xi_{I,\vec{\tau}} := \prod_{I' \in pred(I)} v^{\ddagger}_{I,\vec{q},\vec{u}}(\vec{\tau},\sigma_{I,\vec{\tau}})^{weight(I',I)}$$

with the earliest time of processor allocation to I being

$$\sigma_{I,\vec{\tau}} := \min\{t \in \mathbb{GT} : \exists P \in \mathbb{P} : \tau_{I,P}(t) > 0\}$$

The definition of the value function $v^{\bullet}$ then reduces to

$$v^{\bullet}_{I,\vec{q},\vec{u}}(\vec{\tau},t) := \xi_{I,\vec{\tau}} \cdot v_{I,\vec{q},\vec{u}}(\vec{\tau},t)$$

This is in fact the definition of value functions which we used for the experimental work.

4.5 Dynamic Scheduling Scheme

From the static task graph a partial task instance graph is derived and regularly updated at runtime. Given a resource allocation for a specific task node, a heuristic optimisation algorithm tries to achieve the highest possible value for the task under consideration by evaluating the child nodes' value functions. From this description it seems obvious that value functions are to be calculated in a bottom-up manner. However, as we do not need to evaluate value functions for all parameter settings, it is far more efficient to calculate them in top-down manner and apply lazy evaluation techniques to avoid unnecessary computations.

Figure 4.21 shows the main components of the implemented scheduling architecture.

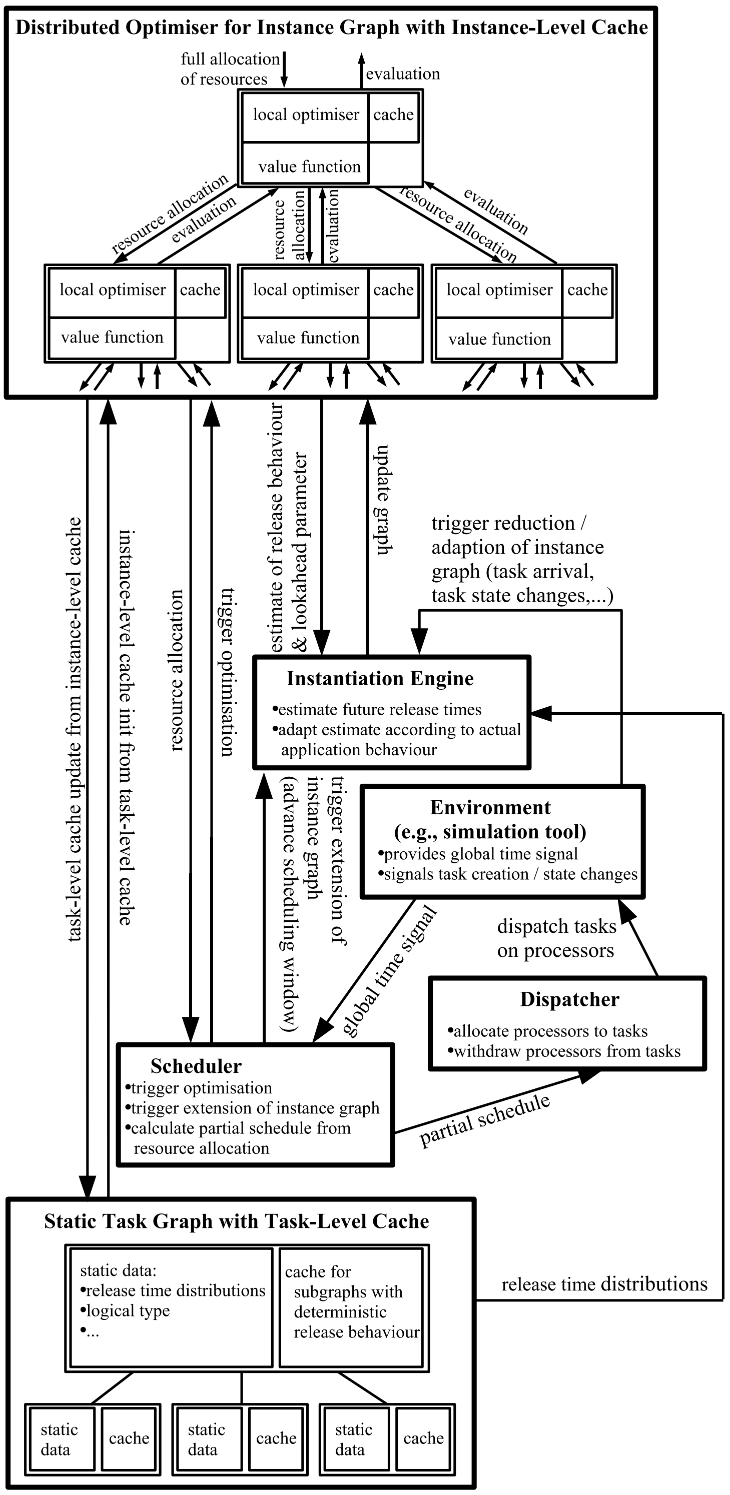

Distributed Optimiser for Instance Graph with Instance-Level Cache
full allocation of resources
evaluation
local optimiser
cache
value function
resource allocation
evaluation
resource allocation
evaluation
resource allocation
evaluation
task-level cache update from instance-level cache
instance-level cache init from task-level cache
resource allocation
trigger optimisation
estimate of release behaviour & lookahead parameter
update graph
trigger reduction / adaption of instance graph (task arrival, task state changes,...)
Instantiation Engine
•estimate future release times
•adapt estimate according to actual application behaviour
trigger extension of instance graph (advance scheduling window)
Environment (e.g., simulation tool)
•provides global time signal
•signals task creation / state changes
global time signal
dispatch tasks on processors
Dispatcher
•allocate processors to tasks
•withdraw processors from tasks
Scheduler
•trigger optimisation
•trigger extension of instance graph
•calculate partial schedule from resource allocation
partial schedule
Static Task Graph with Task-Level Cache
static data:
•release time distributions
•logical type
•...
cache for subgraphs with deterministic release behaviour
release time distributions
static data
cache

Figure 4.21: Overview of the scheduling architecture

Static task graph: The static task graph is the problem description provided by the application engineer for the use by other components. The main data stored in the problem description are the structure of the task set, its use of the method library, quality and utility functions, release time distribution, periodicity classifications with additional data (e.g., period length), logical type of tasks and the dependency graph. Caches are kept with individual nodes of the graph to store resource allocations for subgraphs with deterministic release behaviour that have been calculated so far. These allocations can be reused when the same subgraph is instantiated for the next time.

Instance graph and allocation optimiser: At runtime, an instance graph is gradually constructed from the static task graph. The root node is assigned the entire set of processors at all times, and this resource allocation is passed on to the children. At each node, local optimisers with a local value function are available. The optimisers use one of the techniques described earlier to calculate suitable allocations of processors to task and method instances. Solutions are stored in local caches for reuse in later optimisation runs on the same instances. These local caches can be initialised from the static cache if there is information available. On the other hand, the content of instance-level caches can be used to update task-level caches.

Scheduler: The scheduler object triggers new optimisation runs calculating resource distributions for the current time window within the task instance graph as well as updates to the instance graph to be performed by the instantiation engine. Finally, the scheduler calculates sequential lists of task assignments for the individual processors (schedules) from a given resource allocation.

Dispatcher: The dispatcher reads the sequential schedules produced by the scheduler and takes over the role of dispatching instances on specific processors and withdrawing processors from instances.

Instantiation engine: The instantiation engine reads information from the environment and the scheduler needed to adapt the instance graph; furthermore it estimates the number of instances of every task occurring within the current time window. These estimates are used to update the instance graph.

Environment: Interaction with the environment (which in our case is a simulation tool) must ensure that an external global time signal is available to drive the scheduling process. Furthermore, actual task releases and state changes must be signaled by the environment.

On the other hand, the dispatcher signals the decisions of the optimisation and scheduling procedures to the environment in the form of dispatch and withdraw actions.

Chapter 5

The Cost of Scheduling and Adaptivity

Never let the future disturb you. You will meet it, if you have to, with the same weapons of reason which today arm you against the present.

Marcus Aurelius Antoninus
[Meditations]

The main advantage of dynamic scheduling over static schemes is that the scheduler can adapt to changing requirements of the application environment. However, we have so far not made extensive use of this ability other than a regular estimation of the future set of ready task instances. The potential for adaptation to changing environments, which makes dynamic scheduling applicable to problem settings with partly unknown behaviour of a system, has to be paid for by an increased run-time overhead. In this chapter we will extend our model of schedulers by taking into account the cost of scheduling (which has previously been neglected) and then adding a feedback component aimed at dynamically determining an appropriate distribution of computational resources between the scheduling algorithm on the one hand (the *scheduling allowance*) and the application program(s) on the other hand (the *application allowance*). This feedback mechanism is part of a so-called *meta scheduler* which schedules the activities of the (original) scheduler. The existence of a feedback mechanism implies that the meta-scheduler operates in a *closed-loop* setting, results of prior decisions influencing future decisions, as opposed to the *open-loop* setting we discussed in the preceding chapters, where this is not the case.

5.1 Adaptive Scheduling

For many applications, service requirements vary over time, and in many cases arrival of tasks happens unpredictably. In adaptive real-time systems, resource needs of applications are usually highly data-dependent and time-variant. Under these circumstances, schedulers must be able to maintain a minimum of service at high load or overload while not wasting resources during times of low load. Adaptive scheduling schemes take into account these dynamic requirements and try to use information on the current load of the system to parameterise the scheduling algorithm or even switch to a different algorithm.

Adaptive scheduling schemes are classified by Lu and Stankovic ([LSTS99]) according to whether they adapt to the current situation either by looking at the environment parameters only (e.g., workload, task release rates, maximal execution times, etc.) or by including the consequences of previous scheduling decisions in making new decisions (e.g., utilisation, deadline miss ratio, actual release times, etc.). The former kind of adaptive scheduling schemes is called *open-loop*, the other one *closed-loop*, as it needs some sort of feedback mechanism. Figure 5.1 shows the basic architecture of these two classes of schedulers.

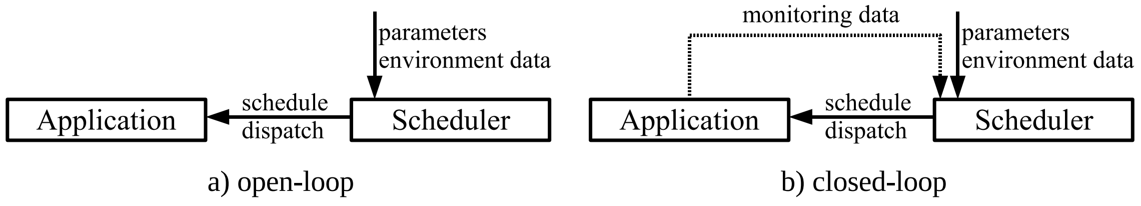


Figure 5.1: Adaptive scheduling schemes

Open-loop adaptive schedulers are basically rather simple heuristics, as they cannot know by construction how the system reacts to their decisions. The performance of such schemes depends largely on the accuracy of the model the designer devises for the actual problem. We will introduce a small set of such heuristics for decision-making in an open-loop quality / utility scheduler later in this chapter.

The more interesting class of schedulers, however, allows data generated at runtime as a consequence of scheduling decisions to be fed back to the scheduler, so that it may learn to modify its behaviour and make different decisions for similar situations in the future if that seems reasonable from the observations. In a later chapter, we will describe the work on closed-loop scheduling found in literature that influenced the development of our own adaptive scheduling scheme in detail.

5.2 The Scheduling / Execution Problem

Dynamic processor scheduling schemes have to address one fundamental problem when they are to be executed on the same processor(s) as the application tasks. Many algorithms assume their own execution time to be short and negligible compared to the tasks to be scheduled. For many simple heuristics, this assumption may be reasonable. However, for the decisively more complex value-based algorithms we use, the computational effort to calculate schedules cannot be ignored.

Like many other problem solving methods incorporating optimisation algorithms, value-based scheduling algorithms can be attributed a tradeoff between the execution time and the quality of the results.

Assume for simplicity that scheduling and execution of partial schedules take place strictly alternating on a single processor and rescheduling is not necessary before the end of the window. Then the order of action is

$$schedulerPhase_1 \rightarrow partialSchedule_1 \rightarrow schedulerPhase_2 \rightarrow partialSchedule_2 \rightarrow \ldots$$

and the sum of the lengths of each scheduling phase and its corresponding partial schedule equals the window size ws:

$$|schedulerPhase_i| + |partialSchedule_i| = ws$$

For a relative scheduling allowance (percentage of processor time awarded to the scheduling algorithm) of $sa \in [0;1]$, we then know that $|schedulerPhase_i| = sa \cdot ws$ and $|partialSchedule_i| = (1 - sa) \cdot ws$.

The i-th scheduler phase runs from time $(i-1) \cdot ws$ until $(i-1) \cdot ws + sa \cdot ws = (i-1+sa) \cdot ws$, the i-th partial schedule from time $(i - 1 + sa) \cdot ws$ until $i \cdot ws$. We define the value of a partial schedule $\vec{\tau}_i$ for the i-th phase as its value at the end, i.e., at the end of the scheduling window: $v_{\vec{q},\vec{u}}(\vec{\tau}_i, i \cdot ws)$. An optimal schedule for the i-th phase is one with maximal value.

Under above preconditions, the length of the partial schedule can only be increased if the scheduling phase is shortened and vice versa. Consider a set of three tasks T_A, T_B, T_C with release times $r_{T_A} = r_{T_B} = r_{T_C} = 0$ and quality and utility functions

$$q_{T_A}(n) = \begin{cases} 0 & \text{if } n < 4 \\ 1 & \text{if } n \geq 4 \end{cases} \qquad q_{T_B}(n) = \begin{cases} 0 & \text{if } n < 5 \\ 2 & \text{if } n \geq 5 \end{cases} \qquad q_{T_C}(n) = \begin{cases} 0 & \text{if } n < 2 \\ 0.5 & \text{if } n \geq 2 \end{cases}$$

$$u_{T_A}(t) = \begin{cases} 1 & \text{if } t < 11 \\ 0 & \text{if } t \geq 11 \end{cases} \qquad u_{T_B}(t) = \begin{cases} 1 & \text{if } t < 9 \\ 0 & \text{if } t \geq 9 \end{cases} \qquad u_{T_C}(t) = \begin{cases} 1 & \text{if } t < 11 \\ 0 & \text{if } t \geq 11 \end{cases}$$

Let the scheduling window be of size 10 and the value function be

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) = \sum_{T \in \{T_A, T_B, T_C\}} \max_{t' \le t} \big(u_T(t') \cdot q_T(\tau_T(t')) \big).$$

For scheduling allowances of 0, 0.3, 0.5, 0.8, and 1, optimal schedules (with a prefix in each row reserved for the scheduler) are as in the table below. The size of the search space (ss) is calculated with the simple rule that allocating cpu time units to tasks with zero utility does not make sense and should be avoided. If the search algorithm can make 10 steps in unit time (sut), we receive the percentage of the search space which can be visited by the scheduling algorithm in the final row.

sa	optimal schedule										value of optimal schedule	search space size (ss)	$\frac{sa \cdot ws \cdot sut}{ss}$
0	B	B	B	B	B	B	A	A	A	A	3	$3^8 \cdot 2^2 = 26244$	0
0.3	–	–	–	B	B	B	B	B	C	C	2.5	$3^5 \cdot 2^2 = 972$	0.03
0.5	–	–	–	–	–	A	A	A	A	–	1	$3^3 \cdot 2^2 = 108$	0.46
0.8	–	–	–	–	–	–	–	–	C	C	0.5	$3^0 \cdot 2^2 = 4$	20
1	–	–	–	–	–	–	–	–	–	–	0	$3^0 \cdot 2^0 = 1$	100

The scheduler can cover the entire search space in the last two cases. We can deduct that an optimal schedule will be calculated for a scheduling allowance of 0.8 or 1 and with a reasonably high probability can be found for a scheduling allowance of 0.5, but with a considerably smaller probability for a scheduling allowance of 0.3. This is expressed by the suboptimal schedule in row 2 of the following table. Finally, it is extremely unlikely that the (presumably arbitrary) solution found for a scheduling allowance of 0 is anywhere close to optimal; the initial configuration of the search algorithm (which in this cases equals the final solution) might assign the processor to the tasks in round-robin fashion. The last column lists the relative value, i.e., the ratio of the values of the found and the optimal schedule.

sa	schedule										value of found schedule	relative value
0	A	B	C	A	B	C	A	B	C	B	0	0
0.3	–	–	–	A	B	A	A	A	C	C	1.5	0.6
0.5	–	–	–	–	–	A	A	A	A	–	1	1
0.8	–	–	–	–	–	–	–	–	C	C	0.5	1
1	–	–	–	–	–	–	–	–	–	–	0	–

The relative value is highest for partial allocation of the processor time to the scheduler. This ratio is bounded by 0 and 1 and is usually smaller at the extremes of very high and very low scheduling allowance. Note that it is undefined for zero-value optimal schedules. The aim of our meta scheduler is to find a scheduling allowance such that the relative value is 1 or close to 1; however, note that the scheduling allowance should not be chosen too big to allow for the optimal schedule value to be as high as possible. Awarding more of the processing resources to the scheduler may increase the relative value, because the optimisation algorithm of the scheduler can explore a comparatively large portion of the search space. However, the drawback of a high scheduling allowance is that only a small percentage of the resources remains available for the application tasks, so that even the optimal schedule is likely to be of comparatively little value. On the other hand, leaving most of the processor time to the application tasks usually results in higher values for optimal schedules; however, a scheduler with very little resources available to itself (in the extreme case, it may only have enough time to act according to a simple straight-forward heuristic) is unlikely to find schedules anywhere near such an optimum, so that the relative value is probably low. The search for optimal distributions of processing time between the scheduler and the application task is known as the *scheduling / execution problem*. A graphical representation of the values involved can be seen in figure 5.2.

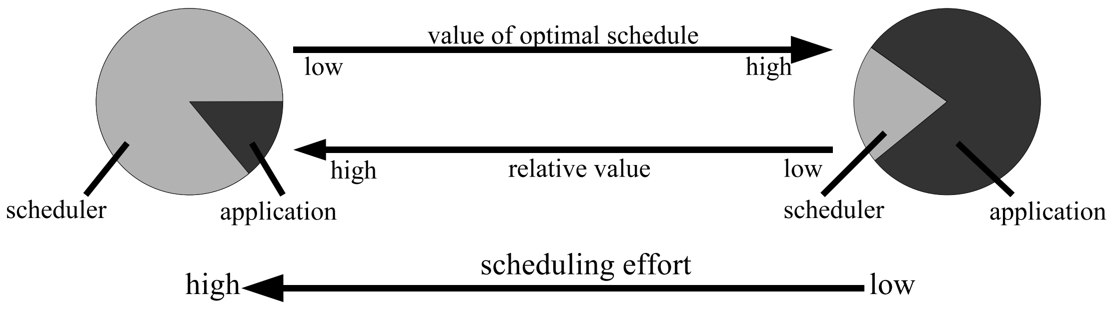


Figure 5.2: Processor time distribution between scheduler and application tasks

The concepts of algorithm selection, anytime algorithms, etc. of the previous chapters can often be applied to the scheduling algorithms themselves, not only to the set of application tasks they work on. Different scheduling algorithms may obviously generate schedules with varying maximum qualities and different quality profiles. The choice of scheduler to calculate a partial schedule at run-time is obviously closely related to algorithm selection, whereas the search techniques forming the core of most of our schedulers in general are anytime algorithms by nature.

For example, a simple heuristic may generate a fairly good result without any relevant effort, but will probably not be able to improve on this solution if given more processing time [1]. On the other hand, a sophisticated scheduling algorithm should be able to calculate close-to-optimal schedules, but may need a certain minimum computational effort before yielding its first preliminary result. At low levels of scheduling effort, schedule value rises monotonically (i.e., the scheduler is an anytime algorithm) with scheduling effort, but starting from a certain level of scheduling effort, the best schedule value (continuous line) differs noticeably from the actual schedule value (dashed line) and starts decreasing at some point. An example diagram for this relationship is shown in figure 5.3.

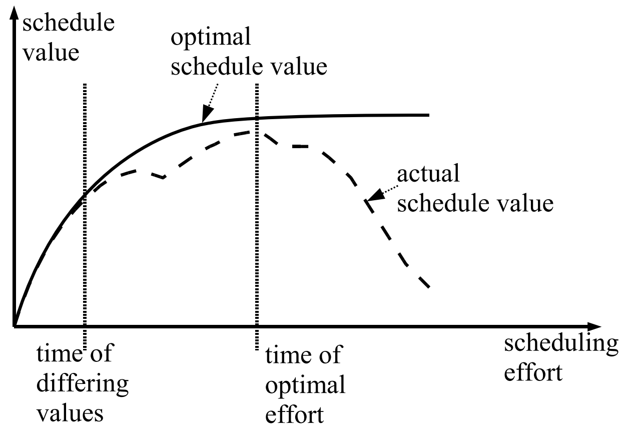


Figure 5.3: Tradeoff between scheduling effort and schedule value

Clearly, this graph for the actual schedule value is not necessarily concave and hence, can well exhibit local maxima. One goal of meta scheduling is the (non-trivial) search for the optimal (or a close-to-optimal) scheduling effort, i.e., a good solution to the scheduling / execution problem. What makes this problem complicated is the fact that the data it works on are in general not available before runtime; only online monitoring reveals the shape of the tradeoff curve. This shape may even be time-variant, so one cannot necessarily rely on the data after an initial learning period.

In the decision-theoretic approach to our specific scheduling problem, it can become highly expensive to cover any sufficiently big subset of possible states even for the near future, especially in cases where transition probabilities to neighbour states are low. The search space for the local search algorithms in the sliding-window approach grows exponentially with the window size[2].

[1]i.e., in an anytime algorithm model, the quality of such heuristics usually degenerates to a two-valued function

[2]in fact, it grows exponentially in the number of elementary intervals, and we assume the average size and hence the frequency and number of elementary intervals within the time window to remain approximately constant over time

Therefore, regardless of whether the designer chooses to generate conditional or unconditional schedules for nondeterministic applications, a scheduler usually has to limit itself to smaller state sets and window sizes than might be desirable and hence may have to perform more or less frequent rescheduling in either case.[3] Hence, in the specific case of the quality / utility scheduling schemes we suggested, the scheduling effort can be adjusted by the window size or state set size parameters as well as algorithm-specific parameters like the cool-down factor of the Simulated Annealing algorithm.

Some research groups have pointed out the necessity of making decisions on whether and when to run a dynamic scheduler. For example, [HAR99] investigated into the tradeoff between the quality of a schedule and the effort required to obtain it. This kind of scheduling scheme needs search algorithms with anytime properties, such that a feasible solution can be obtained whenever the search process is interrupted. Similarly, algorithms have been proposed that bound the scheduling effort a priori, such that interrupting the search process is never required (see section 5.3.1). For simplicity, we assume the scheduler works in consecutive scheduling phases, i.e., instances of the scheduling algorithms whose execution may be interrupted just as any other task, and that schedulers are optimally parallelisable. An even more realistic scheme would consider, e.g., the cost of migration of schedulers between processors and the cost of communication in a parallel scheduler setting. Terms frequently used for the reasoning about the cost of scheduling itself are *meta scheduling* or *deliberation scheduling*, largely depending on the nature of the original application to schedule and the respective research community.

5.3 Meta Scheduling Techniques

This section introduces two of the main approaches to meta or deliberation scheduling. Both of them are described in an open-loop setting. This means that the data on which decisions are made by the meta scheduler are unrelated to prior decisions.

In figure 5.4, the flow of data from the meta scheduler to both the application tasks and the application scheduler is acyclic. The meta scheduler decides on the distribution of resources between application tasks and application scheduler and triggers the actions of the application scheduler on the basis of user-defined parameters and data it receives from the environment, especially on the release of new task instances. The application scheduler then decides on appropriate schedules for the application tasks. The meta scheduler has no knowledge of the consequences of its own decisions and of those of the application scheduler.

[3]Note that due to the factor of nondeterminism, some rescheduling and schedule adaptation would have to take place even if unlimited resources were available.

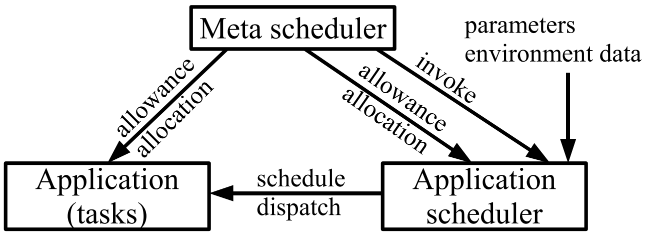


Figure 5.4: Open-loop meta scheduling

In the following section, we will finally extend this model to closed-loop meta scheduling to allow the correlation of consecutive decisions we desire. Meta scheduling should possess the property previously assumed for the application scheduler: the effort needed to make its decisions must be small compared to both application tasks and application scheduler and can be neglected. Otherwise, multi-stage scheduling in a tree structure of schedulers is possible in theory, ultimately requiring only the computational effort for the root scheduler to be small enough for not having to deal with it. However, hardly any evidence of genuine multi-stage scheduling could be found in practice; one of the few examples is the work by Regehr et al.[Reg01].

Two major approaches can be recognised dealing with meta scheduling problems. One of them stems from the area of real-time scheduling, where more complex algorithms applied in dynamic settings have made it necessary in recent years to handle the scheduling effort explicitly; before that, complex algorithms were usually deemed to be applicable only to static scheduling, and the computational overhead of dynamic scheduling was ignored for the biggest part. The second approach was developed in the artificial intelligence community, where reasoning about the cost of making decisions and when to make them (*deliberation scheduling*) has never been uncommon. We note that techniques used in these approaches are very similar despite the differences in their origins and, for this reason, in terminology. We can categorise both of them along the domain of the original scheduling or planning problem. In this thesis, we only deal with problems in the time domain, i.e., real-time scheduling and temporal planning. Other interesting domains like route planning and deliberation scheduling for this kind of applications are outside the scope of this work.

5.3.1 Real-Time Scheduling Approach

A first step towards an explicit handling of the scheduling effort is the predictability of execution times for scheduling algorithms. Zhao, Ramamritham and Stankovic developed a series of scheduling algorithms whose complexity can be bounded in a predictable way

[ZR87, ZRS87a, ZRS87b].

However, knowing upper bounds on the execution time of scheduling algorithms may not always yield the desired fine-granular control over the scheduling effort. For this purpose, Hamidzadeh et al. [HAR99] developed a scheme called Time-Controlled Dynamic Scheduling (TCDS) which is based on iterative construction of partial schedules, such that the scheduling algorithm can be interrupted at any time and always produces a valid partial schedule. In other words, this work is an attempt for an anytime scheduling algorithm and directly addresses the scheduling / execution problem. The dynamic scheduler allows itself a certain portion of the available processor time (the *scheduling quantum*) in each of its invocations (called *scheduling phases*). Instead of estimating the scheduling effort in advance, as was the case in the algorithms by Zhao et al., a TCDS scheduling phase starts extending an initially empty partial schedule until it has used up its quantum it has previously decided upon itself. TCDS does not, however, schedule quality-sensitive tasks; only the scheduling algorithm itself has a notion of quality, defined over three objectives: minimising the number of scheduled tasks which miss their deadlines, the number of tasks not scheduled, and the cost of scheduling. Partially executing tasks (i.e., scheduling tasks which ultimately miss their deadlines) is considered to have more severe consequences than not starting to execute them at all; this kind of deadlines is called *semi-hard* (or, by other authors, *firm*). Each scheduling phase works on a set of currently active tasks; its results are a set of tasks to be scheduled in the near future, a set of tasks predicted to miss their deadlines and therefore prevented from executing, and a new set of active tasks for the next scheduling phase (by removing the tasks just having been scheduled or discarded and adding newly arrived ones). Schedules are extended iteratively by a branch-and-bound method with backtracking. Several heuristics to decide on the scheduling quantum dynamically for each scheduling phase are mentioned in [HAR99]:

- Let the scheduling quantum be lower or equal to the minimum of the slack times of active tasks; this way, the scheduling phase is guaranteed not to cause any task to miss its deadline.

- Decide on a small fixed integer z before runtime; let the scheduling quantum be lower or equal to z times the average task interarrival time; this results in a higher allowance for the scheduling phase if the arrival rate is small (so that it can be anticipated that new decisions have to be made less frequently).

- Let the scheduling quantum be lower or equal to the minimum gained by the two previous rules; each of the two objectives may dominate the decision depending on the current behaviour of the application.

The contribution of TCDS to our work was the notion of scheduling phases deciding on the size of the scheduling quantum first and then starting the actual scheduling algorithm. However, we did not treat the scheduling quantum as a time interval independent of the size of the partial schedule which is computed. We instead introduced the scheduling allowance as the percentage of processing time available for the scheduler in each phase. We thus intended to make the trade-off between the effort for scheduling and the computation time remaining to execute application tasks more transparent. We use the above heuristics for the initial setting of the scheduling allowance, but try to adapt it at runtime according to the consequences perceived. What TCDS lacks is any notion of quality, which is of course a vital component of the objective functions in our problem setting.

5.3.2 Artificial Intelligence Approach

As mentioned earlier, temporal reasoning is usually introduced into artificial intelligence planning systems by specialised real-time subsystems. Whereas most of these models require complete a-priori knowledge of the real-time attributes of the components, Hadad et al. [HKGL03] devised a planning system which is capable of dealing with incomplete knowledge in its real-time component. The architecture of this system consists of independent planning and real-time scheduling agents, where the real-time scheduling agent is responsible for keeping the set of pending actions of the planning agent up-to-date and the planning agent triggers the execution of real-time scheduling actions.

Dean et al. [BD89, DKKN95] pursued extensive investigations on the problem of assigning optimal computational resources to planning algorithms. As their two basic schemes, they distinguish between the *precursor* and the *recurrent* deliberation models. In the precursor deliberation model, a scheduling quantum is set in advance before starting the computation and subsequent execution of the resulting schedule, whereas in the recurrent deliberation model, calculation and execution of schedules are interleaved. The target of planning itself is not necessarily a time-related problem. However, the more general results of artificial intelligence planning research can be applied to temporal planning problems; specifically, the work on deliberation scheduling can complement the ideas of TCDS mentioned above. Obviously, dynamic real-time scheduling requires decisions to be interleaved with execution, so that from the above alternatives only the recurrent deliberation model is applicable. The deliberation scheduler decides on which decision procedure to allocate computational resources at which time by projecting into the future the expected times of occurrence of individual events. The objective function of the overall system is simply the sum of all values of individual activities, and no provision is made for possible interaction of tasks. The goal of the deliberation scheduler is to maximise the objective function, such that the computational resources are optimally distributed among the decision procedures.

Two algorithms were suggested for the deliberation scheduling problem. The first one, called DS-1, requires less knowledge on the shape of performance profiles; only the local values at a specific time are necessary, whereas DS-2 assumes a complete a priori knowledge of the performance profiles. Both algorithms schedule backwards from a given time in the future; whereas DS-1 allocates resources for fixed-size time intervals, DS-2 allocates resources for intervals between two consecutive deadlines. If the necessary information is available, DS-2 is preferable over DS-1, because the additional knowledge of the future behaviour can be used to potentially produce superior results.

The work on deliberation scheduling provided the means for introducing the notion of quality or value into our meta scheduling scheme. The recurrent deliberation model is in fact very similar to the aforementioned TCDS, where the former is described in a more general way and can also be applied to domains other than temporal planning. Apart from the incentives in meta scheduling, DS-1 and DS-2 also contributed to the interval-oriented resource distribution of the sliding-window schedulers in this thesis.

5.3.3 Meta Scheduling for the Quality / Utility Problem

We adopted ideas from both models described above to allow meta scheduling within the framework for quality / utility scheduling and the solution alternatives of chapter 3.

First, we make a few simplifying assumptions to allow for a shorter description of the methodology and also to greatly reduce the implementation effort required. We assume that scheduling algorithms are ideally parallelisable and equally well executable on every processor. This way, the scheduling effort can be distributed uniformly among the processors. Furthermore, the costs of communication between scheduler components running on different processors and of migration of components from one processor to another are neglected. In other words, the hardware architecture appears to the meta scheduler as a single processor insofar as it does not decide on which part of the scheduling algorithm to execute on which processor. [4] Although these may seem rather significant simplifications, these preconditions can be relaxed in a very straight-forward manner. The meta scheduler can be extended to share the same view of the multi-processor architecture as the application scheduler; it need not necessarily distribute the scheduling effort uniformly among the processors, but may instead take into account restrictions on the set of

[4]An alternative simplification would be to require that all scheduling activities take place centrally, i.e., on one processor only.

processors on which the scheduling algorithm (or some of its components) are actually executable, and at which speed. On the same line, communication costs can be taken into account in the same way for the meta scheduler and the application scheduler alike, e.g., by adding constant delays on dataflow dependencies crossing processor boundaries or on migration actions moving the same task or scheduler component from one processor to another.

As in TCDS, we let the meta scheduler decide on its own portion of the available computation time. We use similar heuristics to decide on this scheduling quantum by opting for one of the following:

- Choose the scheduling quantum smaller than or equal to the minimum difference between the remaining time needed for each task to reach a minimum quality and the time remaining for it to retain a minimum utility; this can be seen as a value-based equivalent to the slack time in the traditional real-time parameter set.
- Decide on a small fixed integer z before runtime; let the scheduling quantum be lower or equal to z times the average task interarrival time, just as was the case for TCDS.
- Choose the scheduling quantum smaller than or equal to the minimum of the previous two values.

From the work on deliberation scheduling we used the idea of interleaving scheduling actions and executing application tasks. As we only have complete knowledge of the quality and utility functions of the application tasks, but not of the application scheduler, we were only able to use the DS-1 algorithm to arrange scheduling phases and executions of application tasks on a timeline. Starting from the end of the scheduling window (for the sliding-window scheduling alternative) or the maximum time in the future reached by forward simulation of the state transition graph (for the decision-theoretic scheduling alternative), we reserve certain time intervals for subsequent scheduling phases, so that application tasks cannot be executed during these times. Our aim is to minimise the deteriorating effects of reduced resources available to application tasks. In the given problem setting, DS-1 frequently delays subsequent scheduling phases as much as possible to minimise unaccessible time intervals for tasks in the near future. The reason for not simply opting for the heuristic of maximally delaying scheduling phases is that it can be shown to perform well only if the release times of task instances are known for certain and if the window size or state set is large enough to cover a representative subset of all task instances; for periodic task sets, this would mean to span an interval equalling the least common multiple of all task periods. Note that the basis for the decision on when to award processing time to make scheduling decisions comes from the same objective as in TCDS for modifying the scheduling quantum, namely to minimise interference with application tasks.

To summarise, in the basic meta scheduling scheme for quality / utility scheduling, we adopted ideas from real-time research to decide on a reasonable scheduling quantum and ideas from artificial intelligence planning research to find a suitable timing pattern for the scheduling phases.

5.4 Closed-Loop Meta Scheduling

So far, we have only used data not related to prior decisions for meta scheduling; in this section, we add a feedback component to our model to allow closed-loop meta scheduling.

In our two-stage scheduling model, adaptivity means for the meta scheduler to dynamically decide on an optimal compromise in the effort-quality tradeoff based on the monitoring of the interaction between scheduler and application.

We therefore change the prior meta scheduling model such that data are collected during the execution of application tasks informing the meta scheduler about the consequences of its decisions. Graphically, this is represented by the feedback edge in figure 5.5 received by integrating the open-loop meta scheduling model (figure 5.4) with the closed-loop scheduling scheme (figure 5.1b).

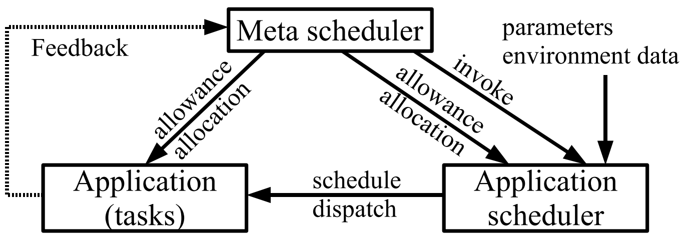


Figure 5.5: Closed-loop meta scheduling

5.4.1 Distributed vs Centralised Adaptation

Two major approaches can be recognised in the research community on adaptive real-time systems. The first one closely links the adaptation of service levels to the specific application and can make use of problem-specific and algorithm-specific parameters as well as knowledge on the available resources to adapt the workload appropriately to maximise overall system performance (figure 5.6a). The second kind of adaptive scheduling mechanisms is situated either at the kernel or middleware level and is performed by system software, namely a quality-of-service manager, a resource manager or a resource kernel (figure 5.6b). It is easier for central system software to monitor all tasks and manage the available resources than it is for distributed agents situated within the individual tasks to share information and coordinate decisions. However, system software can hardly exploit application-specific data for its computations.

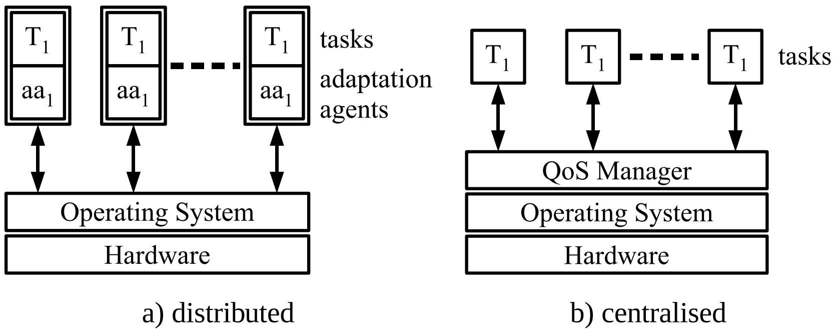


Figure 5.6: Distributed and centralised adaptation

Within the context of this work, quality-of-service managing is an integral component of the value-based scheduler; in this regard, our model is clearly of the centralized class of adaptation schemes.

Several models have been proposed that allow to explicitly modify task parameters in order to facilitate meeting goals. Among these are the elastic bandwidth server by Buttazzo and Abeni [BA02] and heterogeneous computing by Venkataramana and Ranganathan [VR99]. An overview of the information possibly available to adapt schedules and the behaviour of schedulers, although on a more general level than real-time scheduling, is given by Sauer in [Sau99].

These models are, however, outside the scope of the problems we are dealing with, because we consider these parameters (e.g., the period lengths of tasks) to be invariable or at least outside the degrees of freedom for the user or scheduler to decide. The values we concentrate on modifying in our adaptive scheduling scheme are the scheduling allowance and algorithm-specific parameters, but under no circumstances the attributes of the individual tasks making up an application.

5.4.2 Feedback Mechanism for Meta Scheduler

We are going to apply a control-theoretic approach described in [LSTS99] to monitor the performance of the scheduler and adapt the scheduling effort to the behaviour of the environment. A PID controller was chosen in our model due to its simplicity to implement, its applicability despite a lack of an exact description of the future behaviour of a system and its stability for first and second order dynamic systems.

Let $sa_i \in [0; 1]$ denote the scheduling allowance for the i-th scheduling phase. The scheduling allowance is the manipulated variable of the PID controller. For simplicity we again assume the scheduling algorithm can be executed in parallel on the hardware architecture without any additional overhead, though other schemes might be possible depending on the suitability of the hardware and the scheduling algorithm for migration and parallelization. Further, let $\Delta_i \in \mathbb{N}$ denote the length of the i-th phase (scheduler phase and partial schedule). The length of the current partial schedule is generally smaller than the scheduling window or maximum forward simulation time. Instead of introducing a second variable for a controller to operate on, we decided on the following heuristic: the length of the current phase, Δ_i, is the minimum of

- the scheduling window size or the maximum forward simulation time
- the maximum of
  - a user-defined constant minimum partial schedule length Δ_{min}
  - the distance from the start of the current partial schedule to the time of the earliest deviation of the estimated task instance release times from their actual values

We define the start times of the i-th phase as follows:

$$t_1 = 0 \qquad\qquad t_{i+1} = t_i + \Delta_i$$

In other words, the latest time rescheduling can take place is obviously when the precalculated schedule has reached its end. However, if task release times differ from what has previously been estimated, the scheduler is not accurate any more and rescheduling should take place earlier. However, to avoid too frequent rescheduling, a lower bound can be set until which rescheduling is at least deferred. Figure 5.7 shows the case when rescheduling need not be delayed once release times need to be updated, because the minimum partial schedule length has already been reached. In the i-th scheduling window, the scheduler is awarded a computation time of $sa_i \cdot \Delta_i$,

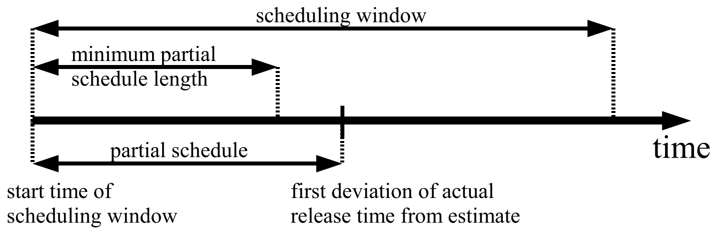

scheduling window
minimum partial schedule length
partial schedule
time
start time of scheduling window
first deviation of actual release time from estimate

Figure 5.7: Scheduling window and partial schedule length

the application tasks receive an allocation of $(1 - sa_i) \cdot \Delta_i$ time units.

As the controlled variable of the controller, we use the change in value density (the slope of the respective value functions) for the root task instance $I^1_{T_0}$ between two consecutive phases, expressed by

$$slope_i := \frac{v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau}, t_{i+1}) - v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau}, t_i)}{\Delta_i}$$

$$\Delta slope_i := slope_i - slope_{i-1}$$

where t_i is the start time of a new scheduling phase (figure 5.8).

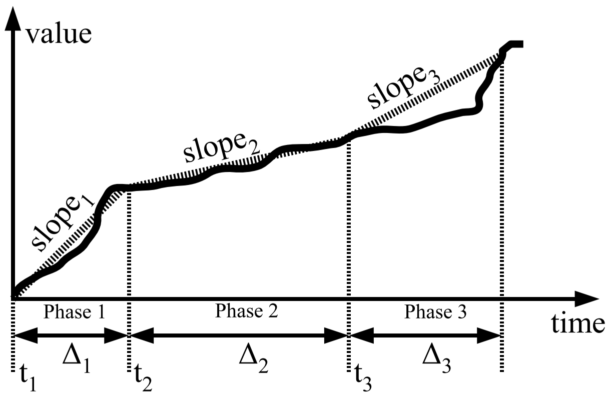


Figure 5.8: Change in value density

Δ_{min} must not be chosen too small; otherwise, the sets of active task instances considered for each partial schedule may not be comparable, and the controller might not be able to stabilise. The system behaviour is monitored for the duration of a phase and the controller parameters are adapted at the beginning of a new phase. This means that the decisions made due to the behaviour of the application in one phase influence the behaviour only for the next phase. It is therefore important that the task set in consecutive intervals is approximately the same in order for the controller parameters to be appropriate. For example, the monitored data for phase 1 in figure 5.9a) result in scheduler parameters which will hopefully be suitable for the second phase, because the task set is similar, albeit not equal. In the example of 5.9b), the task sets in consecutive phases are often hardly comparable, so that it will be difficult for the controller to stabilise; sequences of equal task sets for consecutive phases take turns with consecutive phases with completely different task sets.

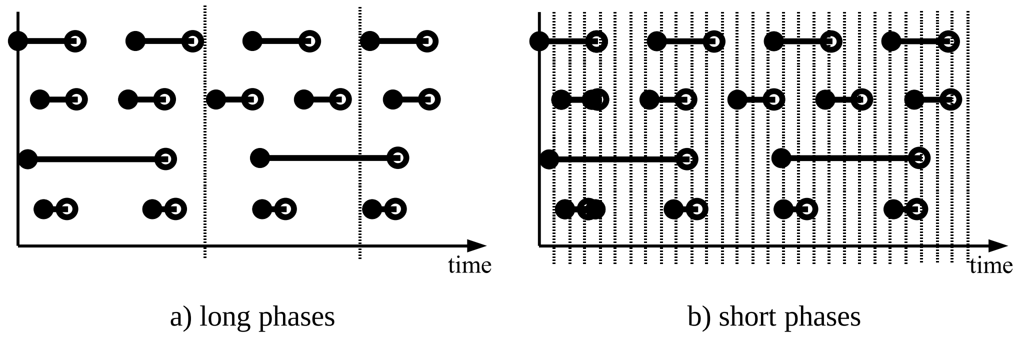


Figure 5.9: Stability for long and short scheduling phases

Obviously, above definition for $\Delta slope_i$ is valid only for $i > 1$. Therefore, as a starting point, we use a different definition for $slope_0$ based on the development of value during the first scheduling phase. For this purpose, we need a parameter $\omega \in]0;1[$ to mark one certain point of the scheduling phase during its lifetime, namely after reaching a certain percentage of its allowance. In analogy to task instances, let τ_{s_i} denote the local time function. For the first invocation of the scheduler, global time equals the local time of the scheduling phase, because no application task can execute before the first partial schedule has been generated. Therefore, the scheduler has exclusive processor access at the beginning, and for $0 \leq t < sa_1 \cdot \Delta_1 : \tau_{s_1}(t) \equiv t$.

We then define $slope_0$ as the terminal gain between the values after $\omega \cdot \Delta_i$ time units of scheduling and the final value of the schedule for the first phase:

$$slope_0 := \frac{v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau}, sa_1 \cdot \Delta_1) - v_{I^1_{T_0},\vec{q},\vec{u}}(\vec{\tau}, \omega \cdot sa_1 \cdot \Delta_1)}{(1-\omega)\Delta_1}$$

The definition for the special case of the first scheduler phase is shown in figure 5.10.

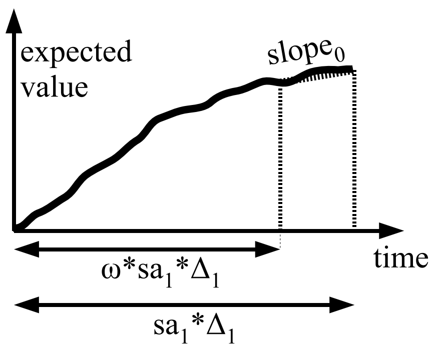


Figure 5.10: Terminal gain

Note that we now do not monitor the actual value of the partial schedule after execution any more, but merely predict the value of the first partial schedule. We could use this definition throughout the runtime of the system, but it seems preferable to apply actual data rather than estimates once these are available. Alternatively, it would be possible to leave the scheduling allowance unchanged for the first two phases, so that adaptation starts later. This would, of course, mean we need not introduce handling for the special case of the first scheduling phase.

There is a tradeoff between the scheduling allowance on the one hand and both the value density and the terminal gain, so that both of these can be approximated by concave functions of the scheduling allowance (compare figure 5.3). It is the aim of the controller to find the optimal scheduling allowance with the slope being approximately 0. Note that the performance profile of the scheduler can have local maxima, but these can usually be left very quickly due to the dynamics of the system and the resulting changes in the profile. Whereas the position of the global maximum stabilises over the course of several phases, the position of local optima does not.

As the set point, we choose a slope of 0, such that the error function for scheduling phase i is

$$err_i := \Delta slope_i.$$

The set point of 0 is chosen due to the following motivation:

- If the slope is negative, too much effort has been spent on scheduling in the preceding phase, taking too much of the computation time from the application tasks; the scheduling allowance should be decreased.
- If the slope is positive, it is likely that an even higher allocation of computational resources could result in even better schedules. To exploit this potential, the scheduling allowance should be increased.

The integral and derivative parts of the controller measure over a distance of $spi \in \mathbb{N}_0$ and $spd \in \mathbb{N}_0$ phase numbers; $C_d \in \mathbb{R}$, $C_p \in \mathbb{R}$ and $C_i \in \mathbb{R}$ are user-defined constants. Error terms err_i are defined to be 0 for negative i.

Finally, we can define the control function as follows:

$$sa_{i+1} = sa_i - C_d \cdot \frac{err_i - err_{i-spd}}{spd} - C_p \cdot err_i - C_i \cdot \sum_{j=0}^{spi} err_{i-j}$$

The initial scheduling allowance, sa_1, must be provided by the user.

To summarise, the feedback mechanism needs the following parameters:

- the initial scheduling allowance sa_1
- the minimum partial schedule length Δ_{min}
- the allowance percentage ω to calculate the terminal gain
- the window size spd for the derivative controller component
- the window size spi for the integral controller component
- the constant factor C_d for the derivative controller component
- the constant factor C_i for the integral controller component
- the constant factor C_p for the proportional controller component

Chapter 6

Case Studies

The path of precept is long, that of example short and effectual.
Seneca

After having introduced a theoretical framework for quality / utility scheduling in the preceding chapters, we will now give evidence of real-world applications which can be modelled very naturally in the scheme we presented. Note that the original problems were taken from literature and are not applications we have been working on ourselves. Our contribution is the systematic modelling and subsumption of these problems under quality / utility scheduling. From the original data available on the specification of these applications, we extract the necessary information to develop suitable task sets with hierarchy and dependency graphs, release specifications, and, of course, quality and utility functions. Having achieved this, we make the original problems accessible to simulation in the environment we will describe in the following chapter. The performance of scheduling algorithms for these applications can be tested prior to deploying them to the original system, which is more difficult with the ad hoc description. The relevance of the general quality / utility scheduling model for describing and simulating existing real-world problems can be shown. We want to point out that dedicated scheduling algorithms for special cases of the general problem may outperform solutions for the general problem, especially for the big classes of applications exhibiting only either quality or timeliness flexibility. However, solutions to the general problem can serve as a basis for competitive analysis.

6.1 Eye Movement Tracking in Laser-Optical Surgery

Fritzsche et al. [FCS+02] describe an application the purpose of which is to keep track of the movements of the eye during a laser surgery and to automatically adjust the laser position appropriately, enabling the surgeon to concentrate on the operation plan without taking care of possible eye movements himself. Before the surgery, pictures of the retina of the patient's eye are taken. These pictures form the basis for an offline map of so-called *landmarks* in the image of the vasculature, i.e., significant features which are unique in their properties and, more importantly, their position relative to each other. At run-time, these landmarks are compared to those found in images taken online. If this comparison succeeds with high probability and within short time limits, the software is able to compute the position of the patient's eye at any time and thus play a vital role in assisting the human surgeon.

The following sections demonstrate the suitability of our quality / utility model to describe this application. Specifically, [SRS+01] and [LSR+02] point out both the quality-flexible and the real-time aspects of the application, which we are going to map into sets of quality and utility functions.

6.1.1 Task Hierarchy

We refer to the main activities to be performed (image grabbing, landmark extraction from this image and landmark-based image matching for spatial referencing [SRS+01]) collectively as *eye movement tracking* and model this part of the application as one main task (eye movement tracking) with three subtasks (image grabbing, landmark extraction and image matching). Eye movement tracking itself is only one part of a bigger surgery assistance system; other components which rely on eye movement tracking are a safety system preventing the laser equipment from harming the patient and a laser positioning system to automatically keep the relative position of laser and eye constant despite possible eye movements. This means an additional hierarchy level with the surgery assistance task as the root node and several subtasks. Furthermore, landmarks are extracted from an image by an iterative algorithm called *vectorisation* or *exploratory tracing*. The exploratory tracing algorithm proceeds in three stages: finding seed points, verifying them and tracing vessels starting from the seed points. Again, these stages are mapped into subtasks of the landmark finding task. The top-down design of the application into a hierarchy of increasingly fine-granular tasks up to a set of atomic computational entities follows very naturally from the original (mostly verbal) description. This way, the entire application can be modelled in terms of a task/subtask tree as defined earlier in this thesis.

The components of the surgery assistance application must be executed regularly; the natural way to simulate this behaviour is by modelling these activities as periodic tasks. Of course, the restriction applies that at most one non-sporadic task exists on each path of the hierarchy graph from the root node to a leaf. Subtask instances are released synchronously with the instances of the parent in our model. This resembles reality better than the frequently used mapping of the application into a set of independent low-level tasks with periodic release specifications, especially when taking into account minor fluctuations in release times (jitter), which usually apply to a series of tasks, not only to a single one.

6.1.2 Dependency Graph

At several levels in the task hierarchy, data dependencies between tasks induce precedence constraints. In our model, this required order of execution is represented by a (value) dependency graph spanned between task nodes. Examples are the orders of execution *image grabbing* → *landmark finding* → *image matching* for the subtasks of the *eye movement tracking* task and *seed point finding* → *seed point verifying* → *seed point tracing* for the subtasks of the *landmark finding* task.

Let us look a little closer at the details of the stages of landmark finding, as they are important for the following discussion of quality flexibility.

In the first stage, a grid is laid across the image and the grid entries are analysed with regard to contrast and brightness levels. A one-dimensional edge-detection operator and local non-maximum suppression are used to find edges in the image. Seed points for the subsequent vectorisation are found by determining spatially close high-contrast edge points and calculating the mid-point between these, assuming that edge points being close to each other belong to the same vessel; of course, this is not always correct.

The second stage starts by filtering from the set of seed points those which can be related to a pair of strong parallel edges; the two strongest edges related to such a point are found by exploring the neighbouring pixels. Thresholds apply to both the strength of these two edges as well as the angle between them: The original point is not considered a seed point for vectorisation if either one of the two edges is not strong enough or the angle between them is beyond a given maximum.

The actual vectorisation takes place in the third stage; starting from a seed point, the two borders of the corresponding vessel are explored separately to determine the direction of the vessel in the neighbourhood of the latest point found. The vessel direction is assumed to cut the directions of the borders into halves. Vessels are traced independently for each seed point; these tracing activities can therefore be implemented by one task each. Once the seed points have been determined, the tracing algorithm works iteratively and the set of tracing tasks can find an increasing number of crossovers and bifurcations with a longer computation time. At any time of the tracing algorithm, an intermediate result is available through the vessel segments and the landmarks found so far. More precise maps can be received by directly building on the intermediate results.

6.1.3 Quality Flexibility

Increasing the computation time of the landmark detection algorithm can increase both the number and the quality of the landmarks and thus improve the likelihood of successful image matching. Landmark sets can be compared to the landmark map computed offline at an early stage. If comparison with a relatively small set of landmarks is successful, computation time can be saved by avoiding to calculate an abundance of redundant additional ones; if comparison is not successful, on the other hand, the partial result can be readily improved upon until the landmark set is sufficient.

Each seed point defines an independent tracing *activity* modelled as a task instance. The problem of assigning computational resources to tracing tasks is a typical value-based scheduling problem, as the distribution of CPU time between these tasks determines to a great extent how well the system performs, i.e., how reliable the landmark matching is. The obvious parameters on which to base the decision of assigning resources to tracing tasks are the strength of the edges, the thickness of the vessel, or a combination thereof. This way one can hope that following thick vessels which show high contrast to their environment will ultimately lead to finding prominent landmarks. The application therefore is quality-flexible, as the accuracy of the spatial referencing depends on the quality of the extracted landmarks, and the number and quality of these features rise monotonically with the computational effort.

Figure 6.1 shows a series of consecutive stages during the landmark extraction algorithm. The set of landmarks is rated according to criteria like number, relevance, or quality. The quality of individual landmarks is calculated from local properties like the thickness and the strength of the segments involved. The overall quality of a landmark map is derived in an additive manner from the qualities of individual landmarks. Although these properties cannot be seen on the series of images, it is obvious that an increasing number of landmarks is found along prominent vessels starting from the seed points.

Blood vessels generally decrease in diameter in the direction of blood flow. In the two-dimensional projection, these vessels appear with diminishing width, and seed points for vectorisation are usually found at the thicker ends. Following the vessel, we are less and less likely to find more prominent landmarks as computation time of the search increases, as both width and contrast of the vessel relative to its environment are components of the quality of crossings and

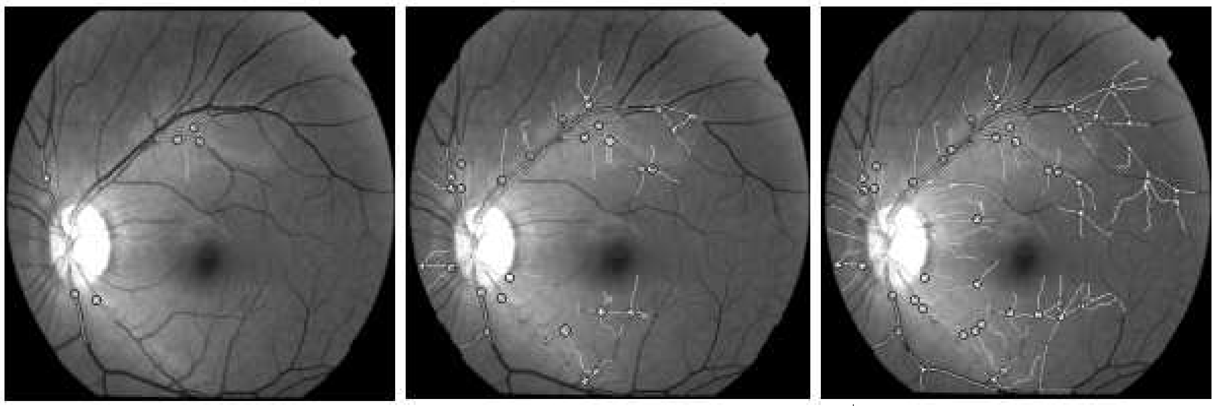
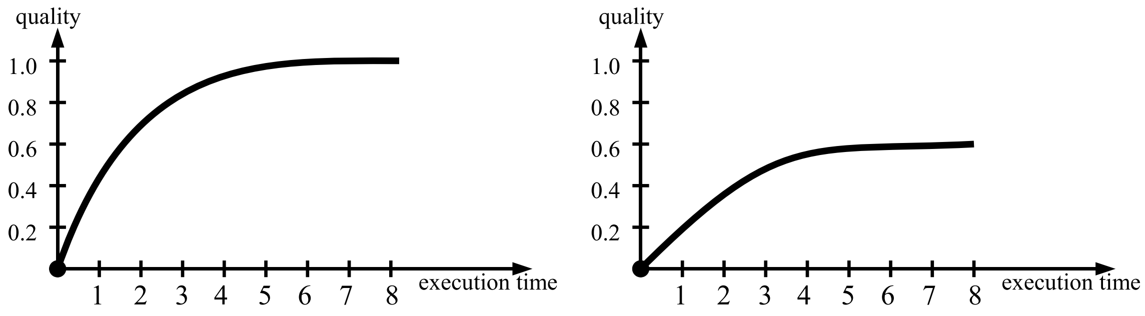

from [SSR+03], reprinted with kind permission from the authors

Figure 6.1: Incremental landmark detection

bifurcations found. We can therefore assume the quality function to be concave and bounded. Furthermore, we assume quality functions for tracing tasks that start from better seeds have a higher maximum quality. See figure 6.2 for example quality functions. The functions have the same shape, but the tracing task for the high-quality seed point (a) can achieve higher quality than the one for the low-quality seed point (b).



a) for task with high-quality seed point b) for task with low-quality seed point

Figure 6.2: Quality functions for tracing tasks

This heuristic definition of quality functions based upon the seed point quality can be adapted at run-time, assigning more appropriate shapes of quality functions to tracing tasks if necessary.

6.1.4 Alternative Tasks

Another standard methodology for landmark extraction, according to [SSR+03], is based on adaptive segmentation of the digital image, skeletonisation to find edges and subsequent branch point analysis or application of interest operators. This approach requires extensive pixel processing, often specialised hardware, scales poorly with image size, and does not provide useful partial results. This is to say that no landmarks can be detected at all before the algorithm has completed, so that vectorisation is usually preferred. However, once the skeletonisation algorithm produces a result, this is usually better than the one received by vectorisation in the same time. Therefore, it is reasonable to opt for the skeletonisation approach when its timely completion can be guaranteed at worst-case conditions, so that its higher accuracy can be exploited. Our quality / utility scheduling model is general enough to include alternative implementations of parts of the application simply by applying different local value functions. In previous chapters we demonstrated the *and*/*or* classification of nodes in the task hierarchy to distinguish between components and alternative implementations of a task.

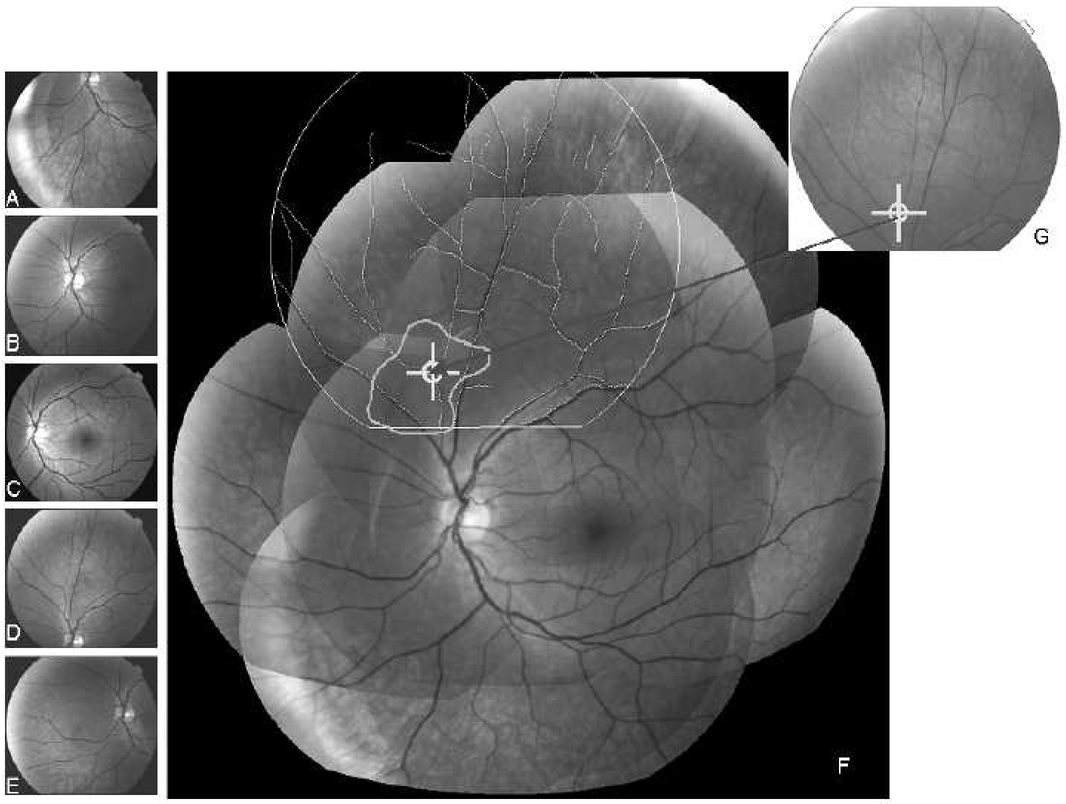
6.1.5 Spatial Referencing

The current position of the eye is determined by spatial referencing of a current picture of the retina to an offline image of the patient's retinal vasculature constructed from pictures taken prior to the surgery (the central big image of figure 6.3F).

This means that the current image is compared to the bigger offline image; if the comparison is successful, the relative position of the current image can be uniquely determined with a high level of confidence. A region within the map of the vasculature is marked according to the surgery plan; the laser is supposed to aim within this area. The current laser position is shown in the most recently taken picture (figure 6.3G) by a cross-hair symbol. Spatial referencing allows to map the cross-hair from the current picture into the offline image and to easily recognise whether the position is within the destination area.

Comparison takes place for a set of characteristic patterns in the images called *landmarks*. Landmarks in the retinal vasculature of the human eye are bifurcations and crossovers of vessels in the projection onto the observation plane.

It is hard to compare individual landmarks, because the properties may vary significantly for different images, one major cause being the viewing angle. Remember the sphere of the retina is mapped onto two-dimensional images. Therefore, the spatial referencing algorithm compares the positions of landmarks relative to each other. To this end, landmarks are grouped together into configurations of two or three landmarks; these configurations are stored in the landmark map calculated offline as “quasi-invariant feature vectors”. At runtime, the landmarks found are indexed, i.e., grouped together according to the same rules that were applied during the generation of the reference map, and subsequently compared to the configurations in the map to find the most similar ones.



from [SRS+01], reprinted with kind permission from the authors

Figure 6.3: Referencing of current picture

The final step in spatial referencing is to try to transform the set of features in the online image into a feature set in the offline map, respecting the neighbourhood relationship. A loop of verification of this transformation and refinement process increases the accuracy of matching. If the transformation finally is sufficiently unambiguous, the position of the online image relative to the offline image can be determined.

6.1.6 Safety Subsystem

One of the subsystems making use of the spatial referencing information is the safety subsystem; the laser positioning system is the other one, but it is too complex to describe here.

As an energy exposure at the wrong location within the eye or for too long a duration at the same location can result in disastrous damages to the patient, it is necessary to determine the exact position of the laser within the area of surgery with only minor tolerable deviations. Specifying the desired location of the laser on an image of the retina taken before the beginning of the surgery, the optical surgeon relies on the assisting technical equipment to track the movements of the patient's eye accurately and adjust the laser position accordingly. However, if the system cannot calculate the position of the laser relative to the eye with very high certainty, it has to signal that it has lost track of the eye movement, and the laser has to be switched off for safety reasons. This measure ought to be a rare exception; system performance degrades with an excessive number of track losses, because switching off the laser increases the duration and the cost of the surgery and decreases the utilisation that can be achieved for the (presumably expensive) equipment.

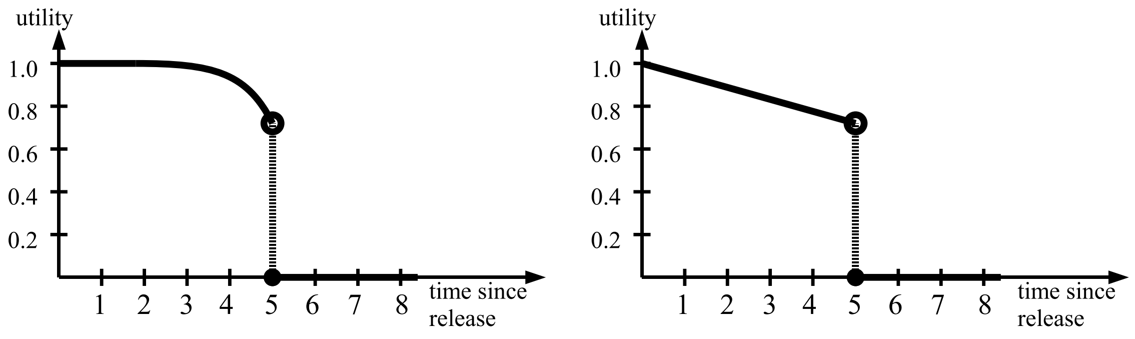
We can identify as the primary goal the reduction of false-negative track loss classifications (the position cannot be determined accurately, but the laser is not switched off). The reduction of false-positive track loss classifications (the laser is switched off although the position could actually be properly calculated) is the secondary goal. The spatial referencing system has to locate the laser position regularly in very small intervals. For simplicity, we assume that this tracking frequency equals the frame grabbing rate of the optical equipment and remains constant during the surgery. The ratio of successful laser locating attempts to the number of images taken (the *locating rate*) as well as the ratio of correct classifications to the number of images (the *classification rate*) depend on factors like the quality of the landmarks in the patient's retina, the extent of eye movement, and, of course, the performance of the tracking application. Ideally, both values are close to 1. In reality, the locating rate cannot be influenced intentionally very much; in general, it is assumed invariable for each individual surgery. A threshold is used to decide on classifications and should be chosen with care to work reasonably on the tradeoff between the two classes of false classifications. Summing up, we receive the diagram of figure 6.4 for the operation modes of the eye movement tracking application.

	track loss detected	no track loss detected
track loss	desired mode in case of track loss, but should be exceptional	false negative: avoid on all accounts for safety reasons
no track loss	false positive: avoid to prevent unnecessarily high costs	normal operating mode

Figure 6.4: Operating modes of eye movement tracking application

6.1.7 Timeliness Flexibility

The application exhibits rather obvious global timing aspects, namely the constraints posed on the spatial referencing algorithm by both the given frame rate and medical parameters. The utility of a classification decreases with increasing time between the time of taking an image and the successful spatial referencing; the frame grabbing rate, though, poses a hard deadline on the computation. The first of these two constraints results in utility functions remaining constantly high up to some critical point when it gradually becomes more dangerous not to make a decision on whether the laser position is still known or not. The second constraint is responsible for the sudden drop in utility; a decision must obviously be made prior to the next image being received (see figure 6.5a)).



a) constant initial phase

b) linear decrease from the beginning

Figure 6.5: Utility functions for tracing tasks

An alternative would be a linear decrease in utility right from the beginning, rendering fast decisions always preferable (see figure 6.5b)). The utility function can either be applied directly to each tracing task or to a common parent node, as all computations belonging to one invocation of the position detection algorithm share the same timing constraints.

The laser can operate in different modes with different levels of energy exposure, influencing the necessary frame rate (decisions have to be made faster for higher energy levels) and thus the utility functions of tasks.

Scheduling for the eye movement tracking application must be done dynamically for a changing set of tasks. The quality of the online images as well as the structure of the vasculature in the current area of interest determines the number of seed points and thus the number of tracing tasks. The time available for spatial referencing may also influence the number of seed points that can be found and hence, again, the number of tracing tasks.

6.1.8 Application Graph

Figure 6.6 shows part of a high-level model of the eye movement tracking application specified in the editor of our simulation environment for scheduling problems (see following chapter).

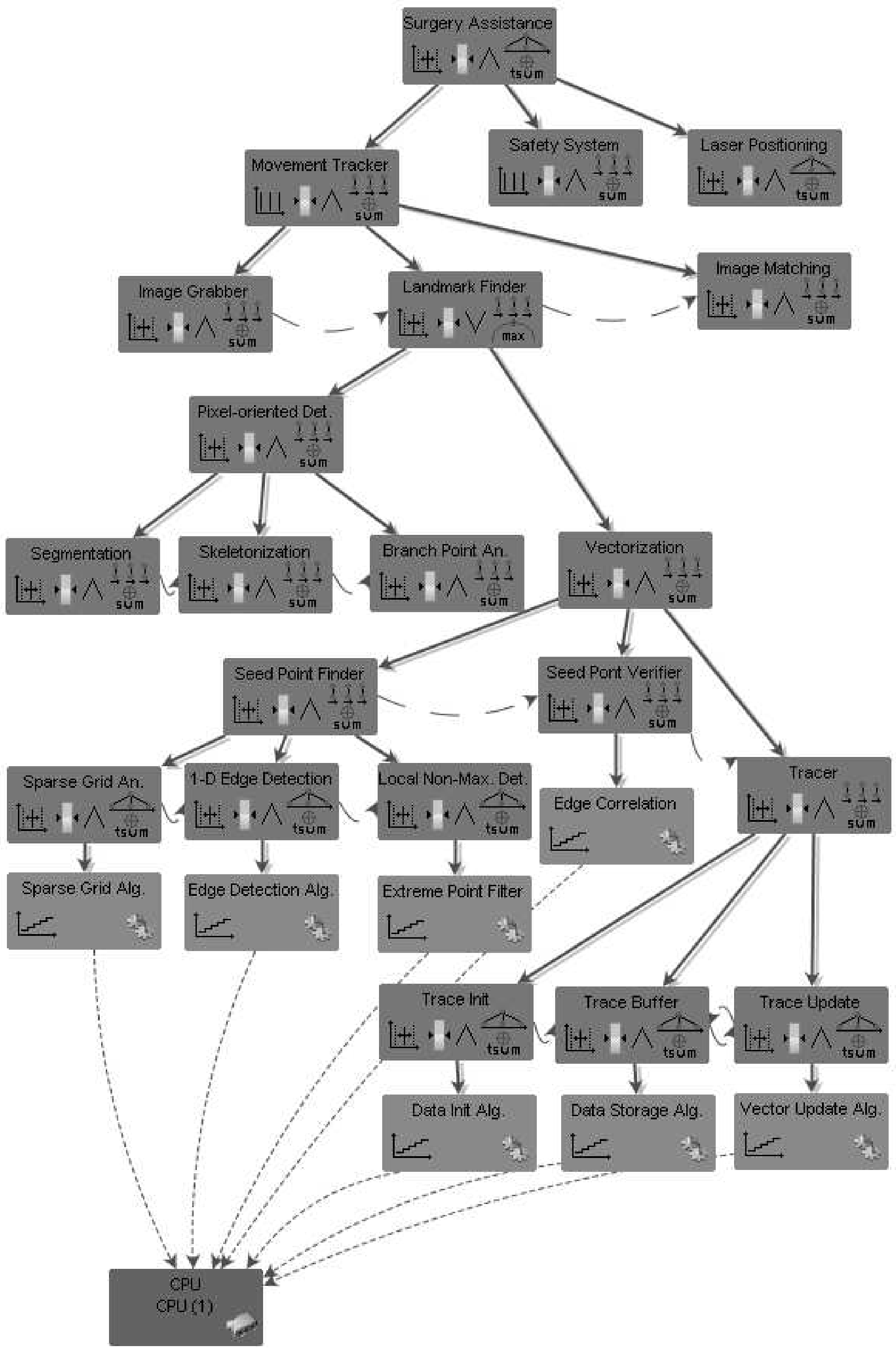


Figure 6.6: Task graph for the surgery assistance application

The graph exhibits all major elements of our model previously described, i.e., a processor at the bottom, method nodes defined to be executable on this type of processor, and a task tree built upon these methods. Note that in this case, the root node of the movement tracking subtree is the only periodic task of the subtree, and that the landmark extraction can be implemented in two different ways, indicated by an *or* type node. The pixel-oriented alternative is only shown in a very high-level description.

The vectorisation task is implemented by a subgraph to find seed points, a filter algorithm to detect false seed points, and an iterative tracing procedure. Remember that tracing is usually performed by a set of activities simultaneously, modelled as individual instances of the tracing task. Instance specifications as well as quality and utility functions are parameters of the nodes of the graph and not shown in the figure. Finally, dependency relationships exist between the phases of the algorithm, such that the qualities achieved by one task may influence the quality that can be reached by others. For example, a poor set of seed points does certainly not foster the hope of finding accurate traces in the digital image.

Remember that the model shown is only a partial description of the eye movement tracking application, and that this application in turn is only part of a bigger surgery assistance system.

6.2 Adaptive Video Streaming Applications

To show that the model introduced in earlier chapters is not only applicable to pure processor scheduling, we will now give examples where the critical resource is in fact primarily network bandwidth.

Streaming video transmission has earned close attention by both academia and industry in recent years. The advances in network technology have made it economically possible for a wider range of users than before to access stored video data over networks and also transmit live video data. While transmission on networks with guaranteed quality of service poses less of a problem, the wide-spread *internet protocol* (IP) and its transport-level protocols such as *user datagram protocol* (UDP) and *transmission control protocol* (TCP) offer no such guarantees. The challenge is the provision of reliable and scalable video streaming techniques on the basis of the best-effort internet services.

6.2.1 Background

In their survey paper [WHZ$^+$01], Weng et al. identify the main issues for real-time transmission of video data over the internet as *bandwidth*, *delay*, and *loss management* and the key elements of streaming video systems as follows:

- video compression
- application-layer QoS control
- continuous media distribution services
- streaming servers
- media synchronisation mechanisms
- protocols for streaming media

Of these, several can be sources of flexibility to adapt to changing environmental conditions, especially fluctuations in bandwidth. Problems occurring due to changing bandwidth are primarily delay and loss of data packets. Both of these may obviously lead to the video appearing unacceptable to the human viewer. Note that in many cases it cannot be determined whether a packet has gone lost or it is only delayed; the effects remain the same in either case.

First, the choice of video compression algorithm has a big influence on the ability of the streaming video system to scale the quality of transmitted data and thus adapt to changing levels of bandwidth in real-time. Obviously, raw video data usually consumes too much of the available bandwidth, making compression schemes a necessary part of any such system. Scalable compression mechanisms like the SPEG (scalable MPEG) extension to MPEG-1 or the FGS (fine granular scalability) encoding of MPEG-4 prioritise blocks of data according to several criteria like frame type (I-, P-, or B-frame) or significance of bits belonging to the layers of the raw data. For example, the compression algorithm may split the original bit stream into two or more layers of base and enhancement data. Each additional layer available to the receiver of the bit stream enables it to improve on the quality of the data encoded in the base layer. The compression scheme may allow for spatial, temporal, and signal-to-noise ratio scalability, meaning the ability to adapt the image size, the frame rate, or the quality of individual frames, respectively. An example for a scalable compression / decompression scheme with discrete cosine transformation is shown in figure 6.7 ([WHZ$^+$01]).

After the transformation, a quantisation module selects part of the information to yield the base layer video stream. The same information is subtracted from the original stream to gain the enhancement layer video stream. The decompression is able to reconstruct a basic version of the original video from the base layer stream alone. If the enhancement layer stream is available, its data can be added to produce an improved version of the video stream.

Using an adaptive compression scheme is a first step to scale the performance of a video streaming application to the available bandwidth. The next one is to establish a control mechanism with the goal of selecting only the most important data to be transmitted at each time

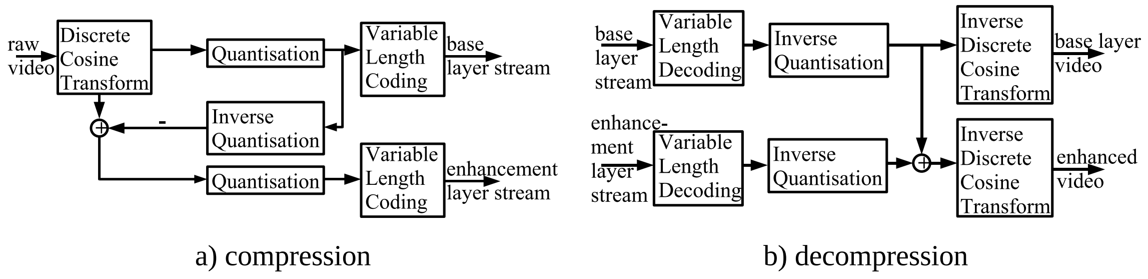


Figure 6.7: Example scalable compression / decompression scheme

instant. Data loss and delay occur when the network is congested; one obvious method to avoid congestion is to adapt the transfer rate of the video stream. The compression algorithm is instructed to produce no more data than estimated to be transferable without congestions. There are *analytic techniques* to decide on the transfer rate based on the packet size, end-to-end delays for the non-congested network, and the estimated packet loss ratio, as well as *probe-based approaches* scaling the transfer rate according to the loss rate for some predefined probe data. Another way to match the rate of the original video stream with the target transfer rate is by means of one or a series of filters to discard from the video stream the less relevant data. The most obvious alternatives are *frame-dropping filters* and *layer-dropping filters*. While the former eliminate all or part of the frames of a certain type (in ascending order of importance), the latter remove packets belonging to improvement layers of the stream.

Relying on possibly small subsets of the original data to preserve minimum levels of information contained in the original video stream, errors can have much more serious effects. Errors in the transmission can be controlled by means of adding redundant data to allow automatic repair of damaged packets or retransmission of missing packets. Note, however, that adding data is contrary to the original goal of reducing bandwidth need, and that retransmission is only feasible if the maximum allowed delay of data is bigger than the round trip time of the video streaming application (the time needed for the signal of a packet missing to arrive at the sender and the packet to finally arrive at the receiver). Errors can also be concealed by temporal and / or spatial interpolation of neighbouring frames; the applicability, though, depends largely on the content of the video sequences. In the following sections, however, we will simplify the models by mostly not dealing with error handling issues in order to be able to emphasise on the quality and timeliness aspects of a streaming video system. In practice, however, error handling mechanisms are an integral component of such a system.

For the sake of brevity, we will not state any details of the other components of a video streaming application, although they may also exhibit potential for performance adaptation, especially as far as the architecture of the streaming server is concerned. We will instead start to describe two specific models for such an application.

6.2.2 Live Video Streaming Applications

Compared to video streaming from stored data, live video streaming usually has the additional requirement of low latency between the time of data generation at the sender and display of the video at the receiver. For this reason, excessive buffering to cope with bandwidth fluctuations and enhance video quality is not an option in this case. Live video streaming applications generally impose direct timing constraints on the travel time of data between the sender and the receiver. It is this class of live video streaming applications we are going to deal with in the following. Flexible timing constraints as well as varying quality for the video stream make is suitable for quality / utility modelling.

As mentioned before, we can identify two approaches of adapting the transmission rate to the available bandwidth. One of them utilises the properties of the compression scheme, the other one the attributes of the transmitted packets to achieve this goal. We are going to show example models for both schemes.

6.2.3 Filter-Based Adaptation

The Priority Drop or Priority Progress Streaming (PPS) algorithm by Huang et al. [HKWF03] suggests to group the packets of the video stream into disjoint sets of packets with time stamps according to their time relative to the start of the video. These groups are sorted in descending order of priority as prescribed by the scalable compression algorithm. The data prepared in such way can now be used to adapt to bandwidth fluctuations without changing the parameters of the compression algorithm and the target transmission rate. From each time window, the least significant packets are dropped at its end.

6.2.3.1 Description

Figure 6.8 shows an example of the transmission of a video stream under PPS with the bandwidth being bigger than the data rate. The original data are grouped into windows and sorted according to priority. Due to the high bandwidth compared to the data rate, no packets need to be dropped.

This kind of control is very fine-granular and can act promptly upon changes in the available bandwidth, because the packets are dropped at the very moment they are delayed due to network congestion. The scheme is also very easy to implement, readily applicable to video multicast and

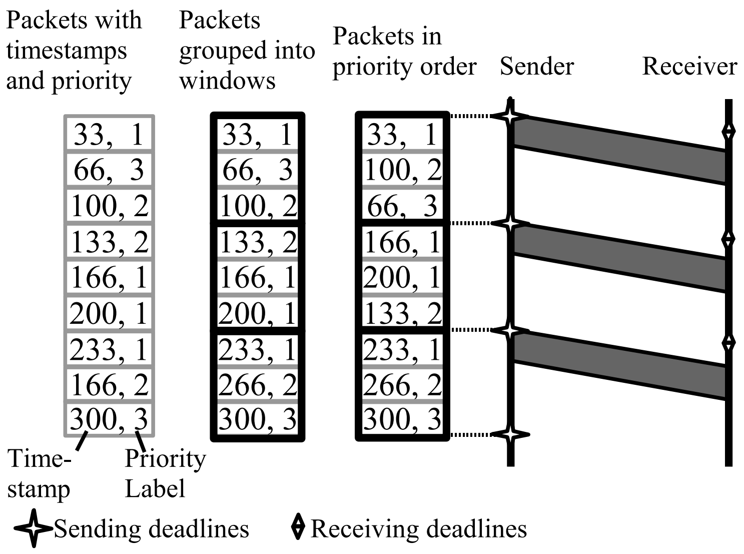


Figure 6.8: Example PPS data stream without delay or congestion

broadcast applications with possibly different transfer rates to each receiver and does not require any form of feedback information from the receiver.

The problem of this scheme is the additional latency introduced by reordering packets in a time window. Furthermore, the Priority Drop algorithm may not always produce good results for video transmission, especially if the data rate at the sender is much higher than the available bandwidth. A compression algorithm aware of at least the order of magnitude of the target transfer rate may perform more elaborate computations to reduce the size of data than an algorithm completely unaware of this information.

6.2.3.2 Flexibility

If the data rate exceeds the available bandwidth, the network may become congested. Such a situation is depicted in figure 6.9.

Some packets of the third window cannot be transferred and must be dropped. In terms of quality / utility scheduling, the transfer of video data cannot be achieved with the highest quality.

On the other hand, PPS deals with increased delays by putting off the receiving deadlines. As this incurs temporary annoying pauses in the video display, it is certainly not a desired effect and should be penalised. In the quality / utility scheduling model, this can easily be done by stating that the transfer does not operate at its highest utility level any more. Note that PPS deadlines are not hard, but rather flexible ones; otherwise, shifting them in the way proposed would not be possible. Figure 6.10 shows the case of receiving deadlines being missed and put off.

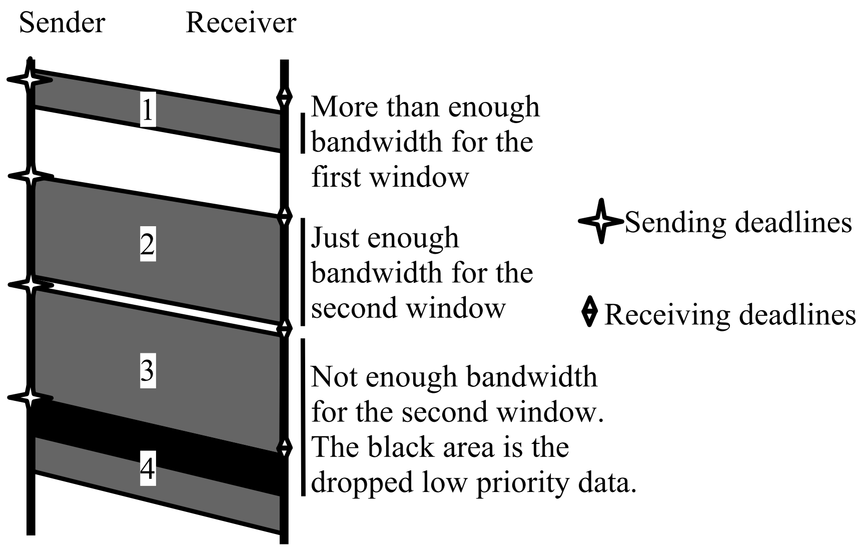

Sender
Receiver
1
2
3
4
More than enough bandwidth for the first window
Just enough bandwidth for the second window
Not enough bandwidth for the second window. The black area is the dropped low priority data.
Sending deadlines
Receiving deadlines

Figure 6.9: Example PPS data stream with congestion

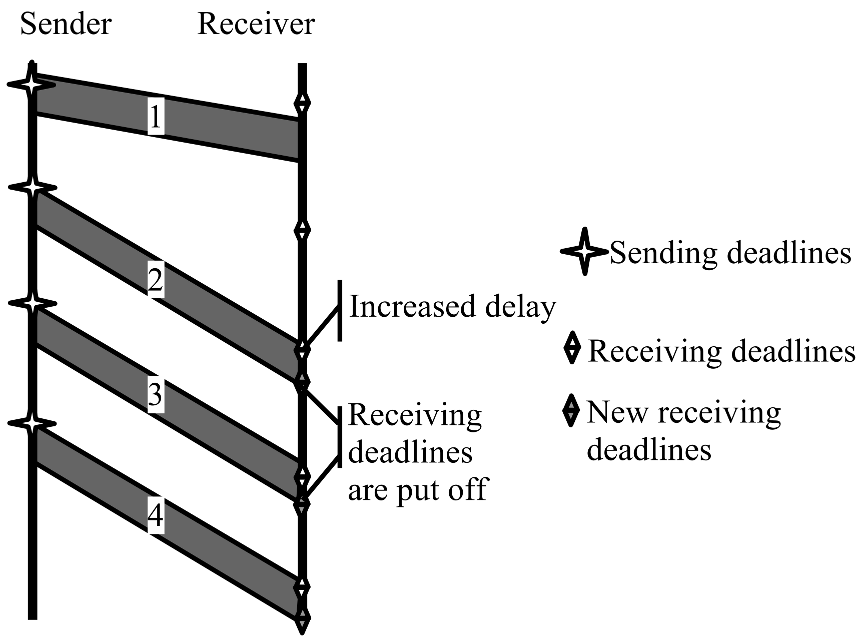


Figure 6.10: Example PPS data stream with delay

6.2.3.3 Application Graph

Figure 6.11 is a model of the PPS live streaming video system. The sender compresses the raw data before grouping the packets into time windows and sorting them according to priority. The system tries to transmit all packets via the TCP connection, but discards those whose time win-

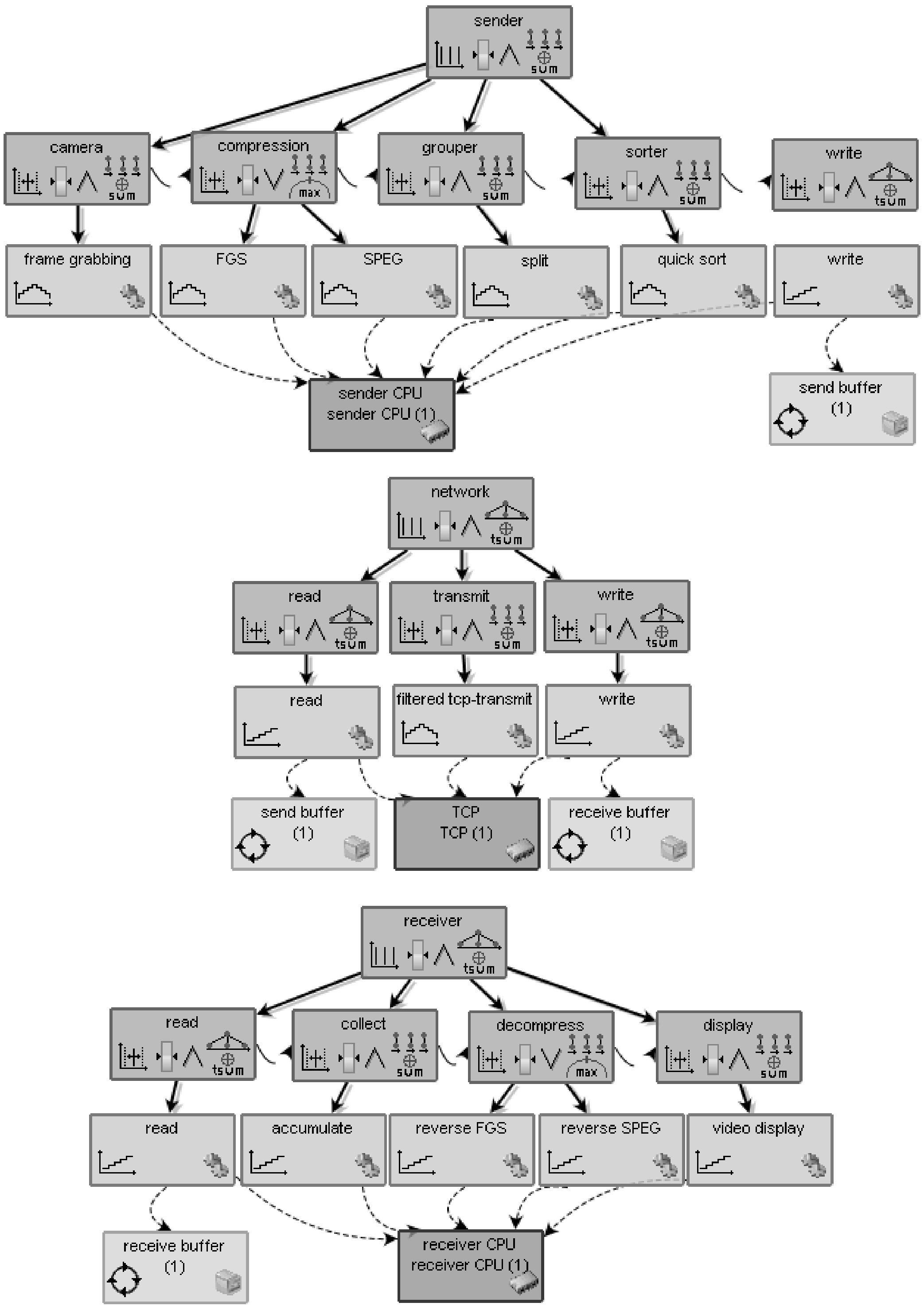
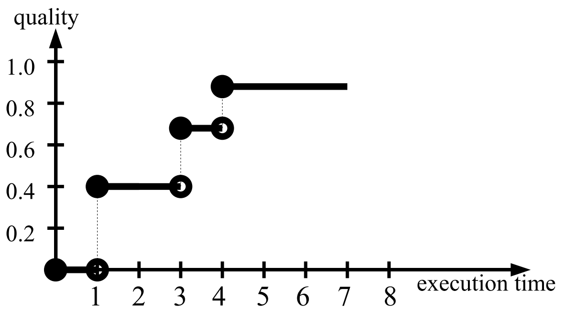
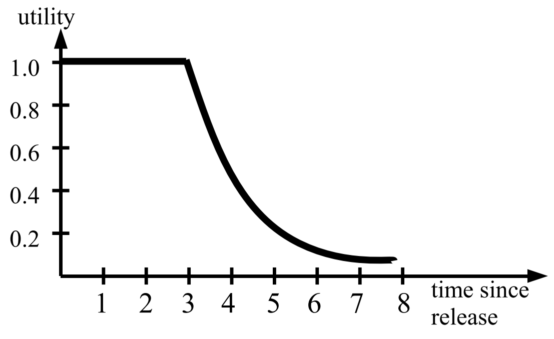


Figure 6.11: PPS streaming video application

dows have already reached their ends. The receiver collects the packets arriving from the network, uncompresses them and feeds them into the display module. The communication is strictly unidirectional, and the sender does not receive any feedback on the status of the network or the quality of the video display at the receiver. Buffered communication as in this example was not considered in the scheduling algorithms described earlier. However, is does not incur additional constraints if we can assume the buffers do not under- or overflow, which should be the case during normal operation of the video stream. Note that the application consists of three components; we cannot apply centralised scheduling to the components of this distributed application. Finally, note that the resource access edges for the buffers do not convey the direction of dataflow, which can be graphically represented in the design tool only for synchronised communication.



a) quality function for TCP transmit



b) utility function for network root task

Figure 6.12: Quality and utility functions for PPS tasks

The quality function of the transmit method is value-discrete, as there is only a finite number of adaptation levels possible according to the number of packets of each window that can be transmitted successfully (figure 6.12a)). However, the execution times of all methods on both the sender and the receiver CPU can be assumed to be constant; therefore, their quality functions only have two possible values. The utility function for the root node of the network tree measures the timeliness of the data arrival; as there is an initial deadline, it should be assumed that data arriving before this time achieve highest utility. After the deadline the utility decreases; in the original problem description, the deadlines themselves are changed. One possibility for defining an appropriate utility function can be seen in figure 6.12b).

6.2.4 Compression-Based Adaptation

Monitoring and probing the bandwidth at runtime together with an appropriate feedback mechanism allows to tune the parameters of the compression algorithm such that it can achieve the highest possible quality of the video stream for the target bandwidth. Further advantages are fine-granular adaptation and little latency incurred by this method, because packets do not have to be collected and sorted before sending them. One major drawback of this approach is the difficulty to find accurate values for the target transmission rate, because the available bandwidth may change unpredictably, possibly rendering void the assumption of extrapolation of past values into the future. An inaccurate target bandwidth leaves us with the original problems of data loss and delay, the network being either under-utilised or congested. A second drawback is the need for feedback information from the receiver to the sender, resulting in possible inaccuracies due to the delay on the return path; once the information is available at the sender, the circumstances may have changed considerably.

6.2.4.1 Description

The ideas for the following example were taken from the works by Rejaie et al. [RHE99, Rej99] on a mechanism for the adaptation of the target rate called RAP (rate adaptation protocol) and by Liu and Zarki [LZ98] on a similar adaptive source rate control system (ASRC). Both of these are based on the heuristics of additively increasing the rate if more bandwidth is available and multiplicatively decreasing it if the network appears to be congested. This scheme has been shown to provide a good tradeoff between the ability of making use of surplus bandwidth and a fast reaction upon network congestion; it is specifically known that this kind of adaptation converges to a fair share of service between the streaming application and other traffic on the same network. The decision on the target data transmission rate (stated as fine-granular as in bits) is usually made upon parameters like the frame rate, error rate, and buffer content.

6.2.4.2 Flexibility

The flexibility of a video streaming application of the adaptive source rate type with regard to quality lies within the ability of the compression algorithm to target different data rates and the transmission system to work on a network with varying available bandwidths and accordingly varying data rates produced by the sender. Utility can be expressed in exactly the same way as for the filter-based approach.

6.2.4.3 Application Graph

Figure 6.13 shows an example model for the adaptive source rate approach to adaptive streaming video. The main activities of the sender remain unchanged compared to the previous model. The

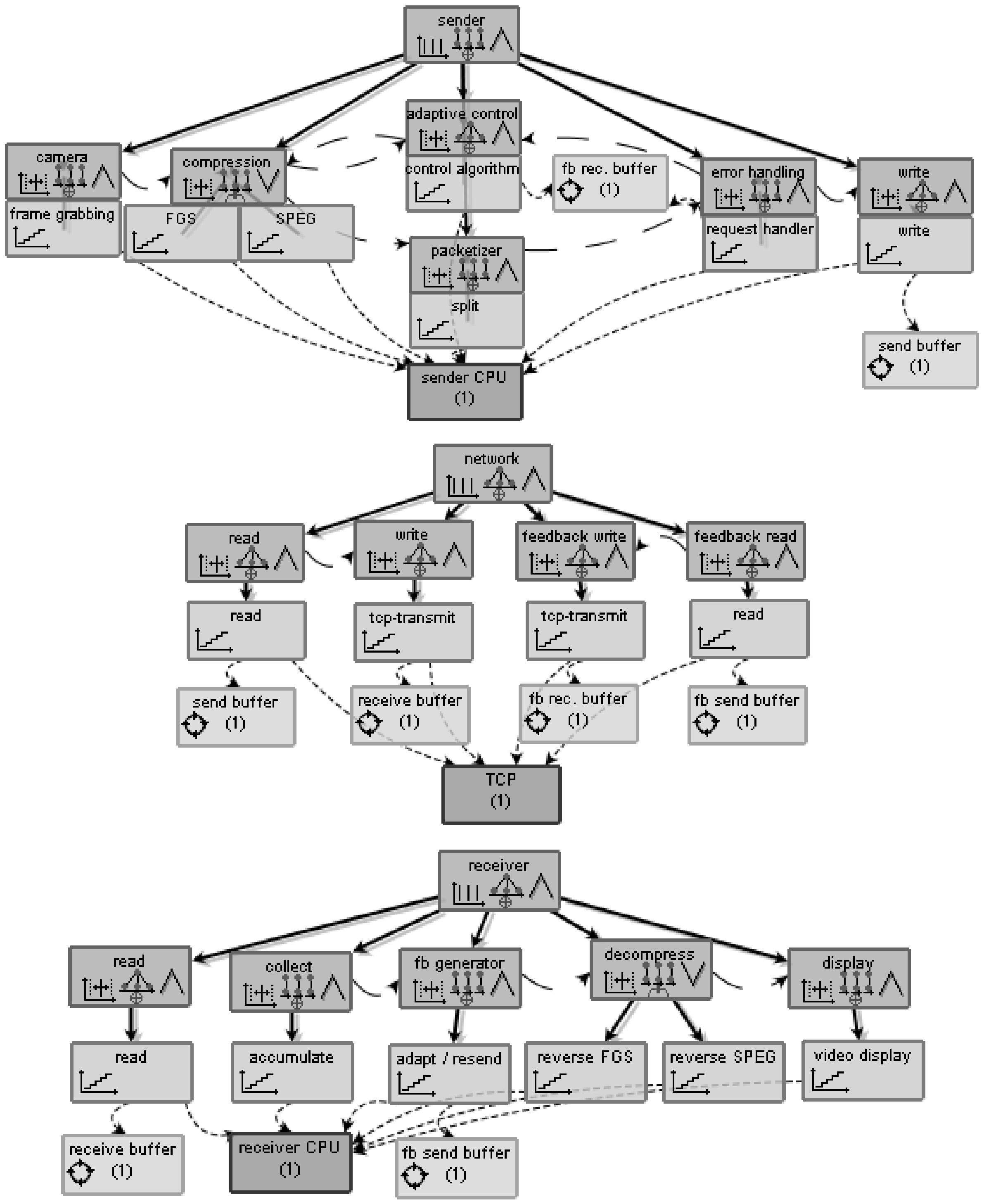


Figure 6.13: Adaptive source rate streaming video application

raw data must be compressed and packetised before being sent via the network connection. However, now it also has to perform two additional tasks, namely explicit error handling and frame rate adaptation. Receiving feedback from the network (signalling congestion) or the receiver (signalling a missing frame or requesting retransmission of packets due to irreparable errors), the sender has to take appropriate action. It may have to adapt the frame rate, reserve some portion of the available bandwidth for retransmission if the network conditions are less reliable, and perform retransmission of the requested packets. The edges from and to the adaptive controller task represent these dependencies. Apart from data collection, decompression and display, the receiver has to check the data for missing packets and errors and produce appropriate feedback messages for the sender. Of course, the feedback messages use the same network resources as the original data. Note that it is necessary to adapt the quality function of the transmit task as a reaction to the adaptation of the target data rate.[1] If the target rate is low, little allocation of the bandwidth of the network should result in a rapid increase of quality (i.e., packets successfully transferred). However, the maximum quality is probably very small, because the data to transfer has deliberately been limited to a minimum. On the other hand, a high target bandwidth means that the transmission can be expected to reach a comparatively high quality (many packets can be transferred successfully), but it may take longer to reach higher quality levels. These two shapes of quality functions are shown in figure 6.14.

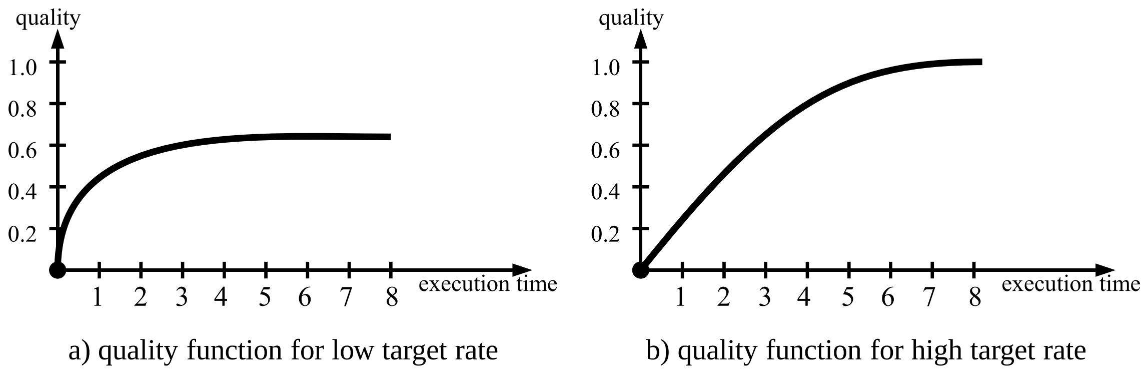


Figure 6.14: Quality functions for transmission task

As can be clearly seen from the quality functions, an inappropriate choice of target bandwidth reduces the transmission quality which can potentially be reached. If the target bandwidth

[1]Changes of the quality function were not explicitly discussed in previous chapters, but pose no problem for our dynamic scheduling algorithms as long as these changes do not occur during the evaluation of the function. This can easily be achieved by constraining changes of this kind to the beginning or the end of scheduling phases. In the previous examples, it was already mentioned that quality functions may be adapted at runtime if deemed appropriate to account for more accurate quality profiles becoming available by monitoring.

is significantly higher than the available one, the transmission task will receive less service than expected and would perform better if the data had been prepared for a smaller target bandwidth. Compare the two quality functions for small execution times, e.g., 1. On the other hand, under-estimation of the bandwidth has a similar effect; compare the functions for a higher execution time, e.g., 5.

Chapter 7

Simulation Environment

Man is a tool-using animal. Without tools he is nothing, with tools he is all.
Thomas Carlyle

An integrated specification and simulation environment called PaSchA (Passau Scheduling Analysis) for scheduling problems was implemented for the purpose of modelling real-time applications and testing the scheduling algorithms of this work and others for such example applications as well as for generic loads. This chapter briefly describes this tool set; more details and a series of screen shots of the graphical user interfaces of the PaSchA components can be found in the appendix.

7.1 Architecture

PaSchA was designed as a set of tools communicating via message-passing mechanisms on the one hand and shared files on the other. An overview of the architecture of the PaSchA system can be seen in figure 7.1.

The core of the tool set is a discrete-time simulator which is capable of applying a wide range of scheduling algorithms to processor and resource scheduling problems with several kinds of timing constraints. Example problems serving as input for the simulator are represented as graph structures and can be generated either by a human application designer using the graphical editor or with the help of a graph generator. The exchange of problem specifications between editor and generators on the one hand and the simulator on the other takes places via so-called application graph files. The other source of input to the simulator is a library of scheduling algorithms

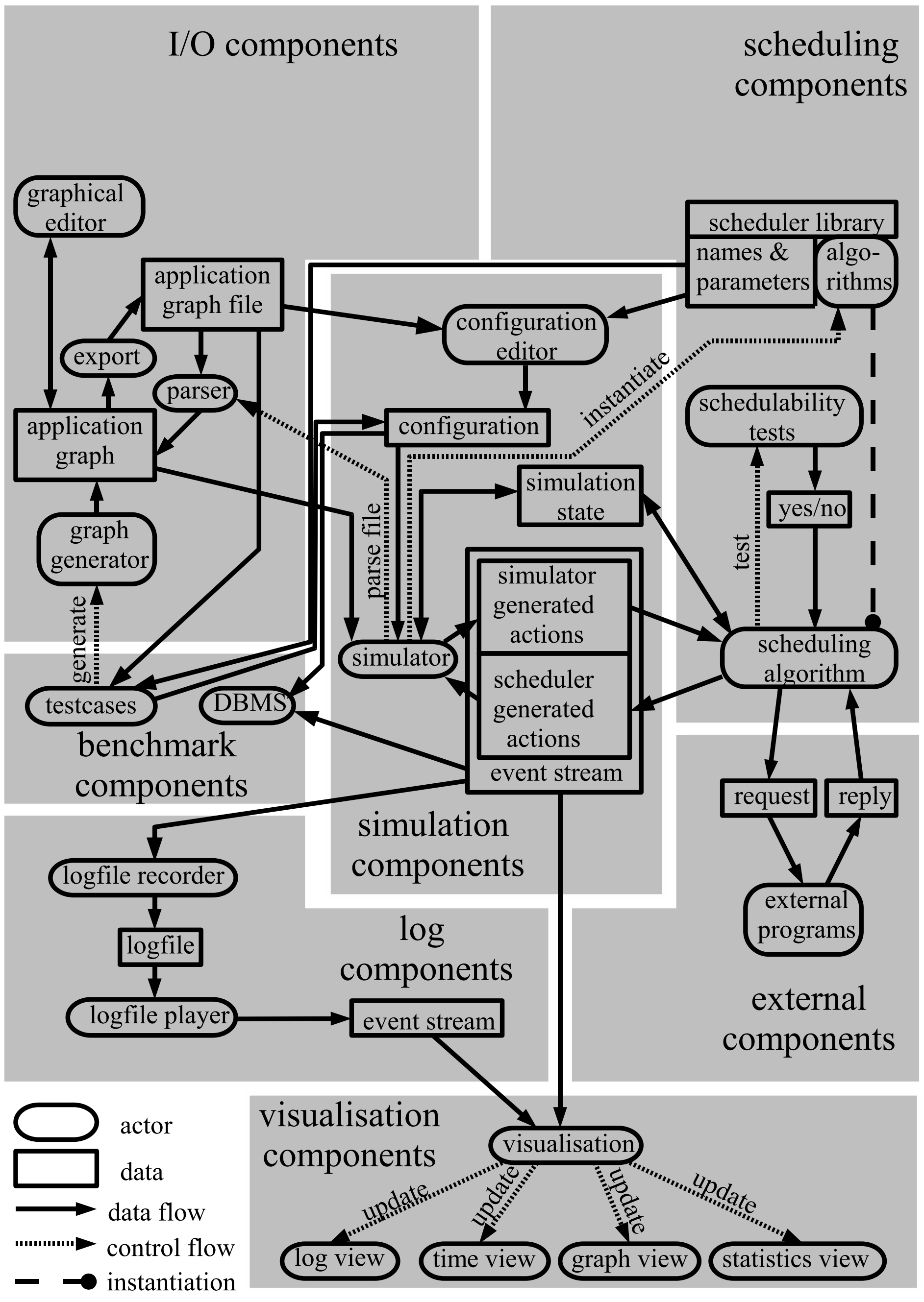


Figure 7.1: Architecture of the PaSchA system

available as Java^{TM} [1] byte code; the library of scheduling algorithms can be easily extended by the user. The interaction between the simulation of the behaviour of an application and the decisions of the scheduler made upon the information provided by the simulator component results

[1]Java is a registered trademark of Sun Microsystems, Inc.

in a stream of events that is passed on to the visualisation components. A series of visualisation modes has been implemented to display the data of interest to the user. Further components of PaSchA facilitate the automatic or semi-automatic derivation of benchmark results for scheduling problems, recording and playing of log files, and interfaces to connect PaSchA to external software tools.

The remainder of this chapter consists of a short section on the model used to specify applications for PaSchA and a more detailed description of the components.

7.2 Application Model

This section serves to give a short overview of the structure of PaSchA application models. A PaSchA application model contains specifications of both a software application and the target hardware architecture on which to execute it. Both of these components are stored within an application graph (7.2). The individual layers consist of sets of nodes spanning several graph structures, both within the layers and crossing the layer boundaries.

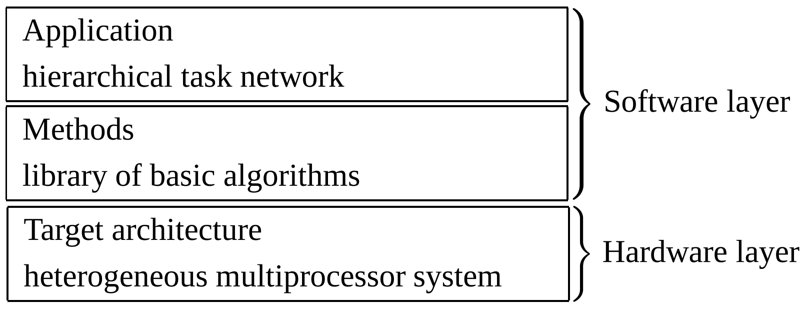

Application
hierarchical task network
Methods
library of basic algorithms
Target architecture
heterogeneous multiprocessor system
Software layer
Hardware layer

Figure 7.2: High-level view of a PaSchA application graph

7.2.1 Hardware Layer

The hardware specification consists of a set of processor types along with an optional set of further resources.

PaSchA allows applications to target heterogeneous multiprocessor architectures as their execution platform. The attributes of processor types give information on how many instances of each processor type are available and which speed modes are defined for this type of processor. Speed modes of processors are assigned a further attribute specifying power consumption; in general, faster speed modes result in higher power consumption.

Non-processor resources contain information on how many units of the resource are available and whether units can be returned to the pool of resources after use or they are consumed and never become available again.

7.2.2 Software Model

The software model comprises two kinds of nodes, namely methods and tasks. Methods can be thought of as basic algorithms available to the application designer as an algorithm library, and an application is built upon this layer of simple algorithms by tasks specifying, e.g., a hierarchy as well as timing and precedence constraints.

7.2.2.1 Methods

Methods are executable on exactly one type of processor. A function mapping time to some scalar value domain correlates the processor time assigned to the method with the value of the overall performance of the application derived from this assignment. This concept generalises the widely used assumption of fixed execution times for components of an application. Obviously, fixed execution times can be modelled very simply. Both the run-to-completion assumption and the anytime execution paradigm have been implemented in PaSchA. In the first case, a stochastic distribution can be specified for the execution time of methods; the simulation component (representing the outside world) decides on an execution time for each method, and the scheduling algorithm cannot influence, but only estimate and monitor the progress of a method. Prior to its completion, a method does not yield any positive value to the application. In contrary to the run-to-completion assumption, under the anytime execution paradigm methods can contribute to the system performance before they have finished. Furthermore, it is the scheduling algorithm which has to determine when to terminate a method. By definition, a method can be executed for an arbitrarily long time in the anytime model. By convention, the function mapping execution time to value should be nondecreasing for all methods. In the PaSchA terminology, methods relying on the scheduler to terminate them are of type *anytime*, methods obeying the run-to-completion assumption are of type *stochastic*, as the simulator decides on the execution time according to some stochastic distribution. Both the time-value function for anytime type methods and the stochastic distribution for run-to-completion methods can be specified in a discrete form as a table of defining points or by coding them directly as Java methods.

7.2.2.2 Tasks

Based upon the library of methods, applications are defined by the application developer as a hierarchical task network consisting of a set of task nodes and two distinct graph structures on them, namely a task hierarchy and a dependency graph.

During simulation, instances of tasks are generated according to the instantiation and release time specifications; instance numbers are assigned to instances to receive a unique order of instances for each task on the one hand and the correct correlation of instances of different tasks on the other. Each task node in the application graph is characterised as being instantiated either *periodically*, *aperiodically*, or *sporadically*. Periodic and aperiodic tasks are instantiated infinitely many times for each parent node instance, whereas sporadic tasks are instantiated exactly once for each parent node instance (or exactly once if this node is the root of the hierarchy graph).

Of the many possibilities to specify stochastic distributions for release times, geometric and uniform distributions were implemented in PaSchA, as they appeared to be sufficient for most models of scheduling problems. Geometric distributions (the discrete equivalent to the memoryless exponential distribution) are mostly used for aperiodic tasks, where it is assumed that instances are released at a given minimum distance in time, but with a probability otherwise remaining constant. Uniform distributions are used for periodic tasks, where release times are known to be within a (usually small) interval of time around the period start (the maximum jitter).

Timing constraints can be modelled in PaSchA in two ways. The first one is by traditional *deadlines* posed on the tasks, i.e., by specifying either in absolute time or relative to the release time of a task instance the latest time when the instance must finish its execution. The second one is the more fine granular specification of *utility* by means of pointwise constant functions of the time passed since the release of a task instance. One other important attribute of tasks is the *quality aggregation function*. This function serves to calculate values representing the task's progress from the values observed for the child nodes. Two classes of quality aggregation functions were implemented in PaSchA. The first one only takes into account child node instances of the same number as the parent, the other one calculates the parent node quality from all child nodes present at a given time. Maximum, minimum, sum, and arithmetic mean are the functions implemented in both of these categories. A value density quality aggregation function as well as an interface for user-defined functions are available.

7.2.2.3 Edges

As mentioned above, two graph structures are defined on the task nodes. The first one is a tree called the *task hierarchy*. In the case of PaSchA, the task hierarchy is an AND/OR tree, so that the subnodes of a task may either represent components (for AND type nodes) or alternative implementations (for OR type nodes) of the parent node. To avoid instance explosion, a phenomenon where an infinite number of instances with an infinite number of children each are generated during simulation, PaSchA does not allow more than one non-sporadic task node along any path from the root node of the hierarchy to a leaf.

The second graph structure defined on the task nodes is the dependency graph, which is directed, but need not necessarily be acyclic or contiguous. Two different kinds of dependency edges were implemented in PaSchA. The first one is called *data dependencies*, which represent hard precedence constraints, i.e., a task (rather, its instances) may not be executed prior to the termination of its predecessor nodes. The other one is called *quality dependencies*, the purpose of which is to indicate that the quality of the target node of the edge is influenced by the quality of the source node. It is not illegal to execute the target node prior to the termination of the source node; usually, the objective function of a scheduler would penalise wrong execution orders. However, it is possible to combine both kinds of dependency edges for the same pair of task nodes. Dependency edges are also allowed between method nodes, where their interpretation is analogous to the one given for task nodes.

Two kinds of edges cross the borders of the layers given in figure 7.2. First, method usage edges connect tasks and methods, where the method node must always be the target node. Note that each leaf node of the task hierarchy graph must use (i.e., be implemented by) at least one method. Processor and resource access edges cross the border from software to hardware layer. Each method is executable on exactly one type of processor, so that accordingly each method node is associated with exactly one processor type node and an optional set of other resource nodes.

7.3 I/O Components

The two ways to create application graphs for PaSchA are the graphical editor and specialised graph generators.

7.3.1 Editor

Here we give a brief description of the functionality of the PaSchA graphical editor; for more details refer to the appendix. The editor implements a multi-document interface, so that the user can work on several application graphs at the same time. The main panel shows one of the graphs at a time, other open graphs can be selected from the tab list. Apart from the usual functions like *copy*, *paste*, *load*, *save*, *undo*, *redo*, and *print*, the editor allows insertion of the different kinds of nodes and edges and opens additional dialogues when necessary to edit their properties. Further features of the editor are zooming (with different sets of important attributes of nodes represented graphically for different zoom levels) and several layout options. Finally, the graphical editor automatically searches for scheduling algorithm classes within its installation directory. An algorithm can be selected and parameterised to test the validity of the application graph for the specific scheduling algorithm. Note that not all graphs that can be specified in PaSchA's general model are suitable for every algorithm. Figure 7.3 shows the main window of the graphical editor with a complete application graph.

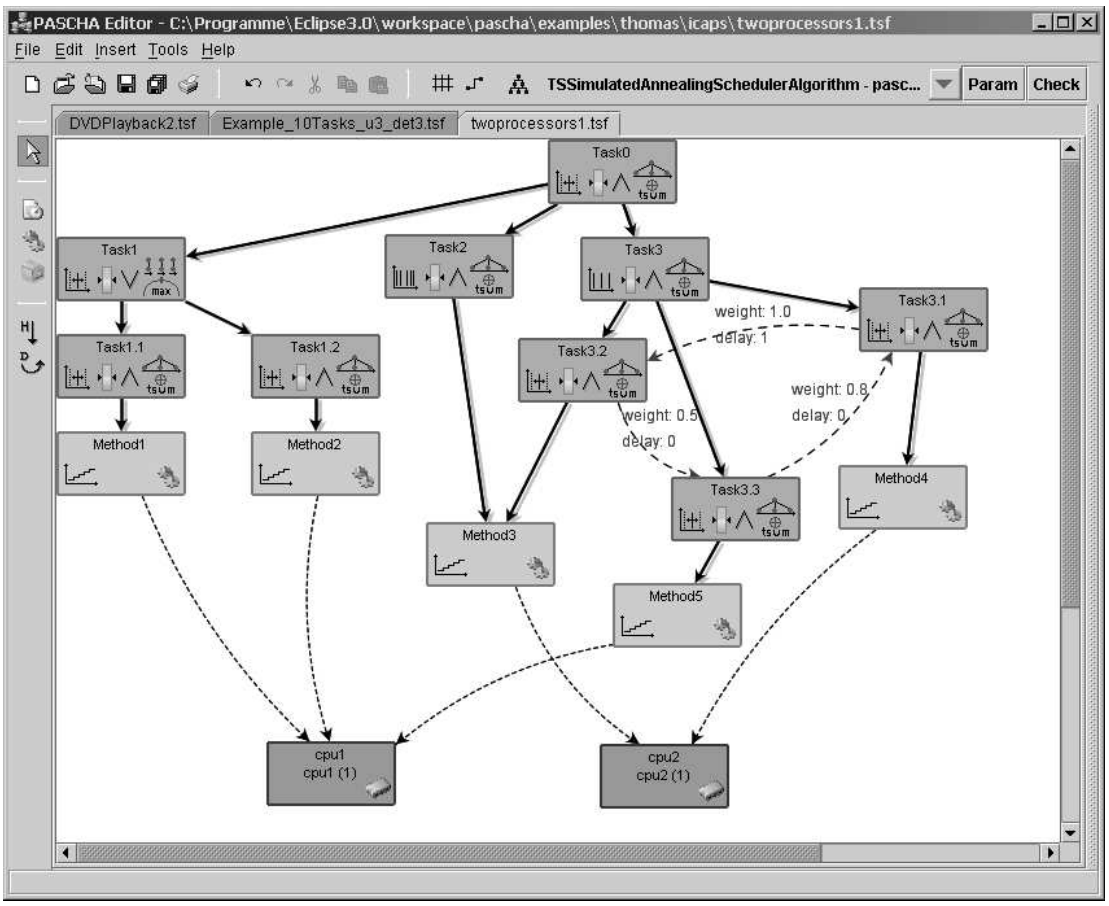


Figure 7.3: Example application graph in the graphical editor

7.3.2 Graph Generators

A second way to create PaSchA application graphs is by using specialised graph generators. This is especially useful for automatic generation of generic example problems with certain characteristics for extensive benchmarking. A graph generator is generally intended to produce reasonable input for one class of scheduling algorithms only. Sets of parameters are provided either via an application programming interface (API) or a graphical user interface (GUI), which the generator algorithm uses to produce suitable example graphs and write them either to the file system or a database. A generator for quality / utility problems is described in the appendix.

7.3.3 Application Graph Files

Application graph files store graphs in XML format and are used for data transfer between the components; the XML scheme also simplifies switching between database and file systems as data repository.

7.4 Simulation Components

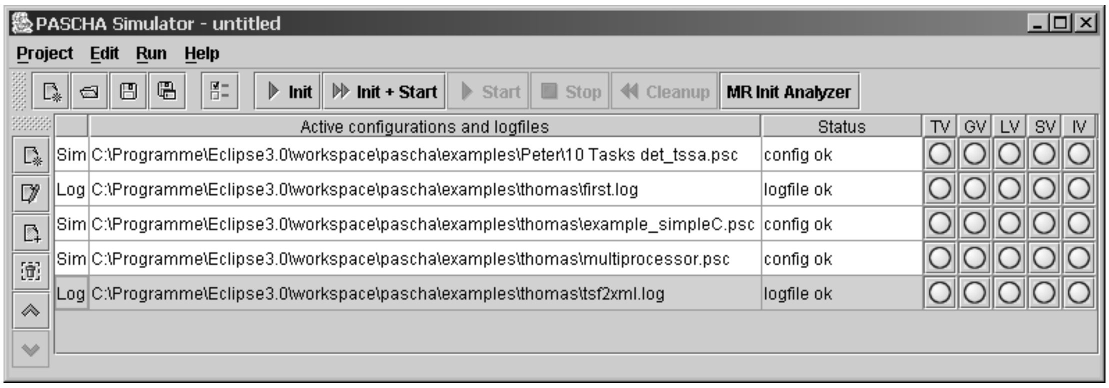
The simulator is the core of the tool set; the actual simulator is accompanied by two auxiliary components: an editor to create configuration files and a graphical user interface to simplify its usage.

7.4.1 Configuration Editor

The configuration editor is a little tool that allows to bind together an application graph with appropriate layout information, a path on the file system to store log files, parameters for the statistics view mode (see section 7.6), a scheduling algorithm, and a set of scheduler-specific parameters into one file. User interfaces to edit scheduler-specific parameters must be provided by the individual scheduler classes. The appropriate scheduler-specific dialogue is opened once the scheduler has been selected from the drop-down list. This list is generated by the configuration editor on the basis of the Java class hierarchy, so that new schedulers become available automatically.

7.4.2 Simulator User Interface

The graphical user interface of the simulation tool (figure 7.4) shows a list of configurations and pre-recorded log files that are to be simulated (in the case of configurations) or played (in the case of log files) simultaneously. Entire lists of configurations and log files can be saved and loaded as play lists within the graphical user interface. Execution of play lists can take place in three modes: as fast as possible, with a minimum delay for each step, or in single-step mode with user interaction. Visualisation modes can be selected separately for each entry in the play list. Simulation of configurations and playing of log files can be started, paused, and stopped via the tool bar of the user interface. The functionality of the simulator is also accessible via calling options for the simulator, which is important for automatic benchmarking.



	Active configurations and logfiles	Status	TV	GV	LV	SV	IV
Sim	C:\Programme\Eclipse3.0\workspace\pascha\examples\Peter\10 Tasks det_tssa.psc	config ok	○	○	○	○	○
Log	C:\Programme\Eclipse3.0\workspace\pascha\examples\thomas\first.log	logfile ok	○	○	○	○	○
Sim	C:\Programme\Eclipse3.0\workspace\pascha\examples\thomas\example_simpleC.psc	config ok	○	○	○	○	○
Sim	C:\Programme\Eclipse3.0\workspace\pascha\examples\thomas\multiprocessor.psc	config ok	○	○	○	○	○
Log	C:\Programme\Eclipse3.0\workspace\pascha\examples\thomas\tsf2xml.log	logfile ok	○	○	○	○	○

Figure 7.4: Graphical user interface of simulator

7.4.3 Simulator

The PaSchA simulator component is time-discrete and communicates with its environment via the file system (configuration editor, graph editor, graph generators, log writer) and message passing mechanisms (visualisation). Its general purpose is to manage the system state for each simulation of a configuration and to provide information necessary for the scheduler to act and for the visualisation modes to display. The most important data the simulator must make available to the outside world are the current time, the release of new task instances, their assignment of resources and computation time, their progress, quality, and, finally, their termination. Furthermore, processor and resource usage are information needed by other components. The states of all task and method instances are determined by finite-state machines. Both schedulers and the simulator itself may only take action on a task or method instance according to the state transition diagrams in figures 7.5 and 7.6.

The main part of the simulator is a (potentially infinite) loop performing the same basic actions at every time step. The three top-level phases of a simulator cycle are performing changes to the system state taking place due to the time progressing, invoking the scheduler, and - after the end of the scheduling phase - interpreting and reacting appropriately upon the decisions made by the scheduler.

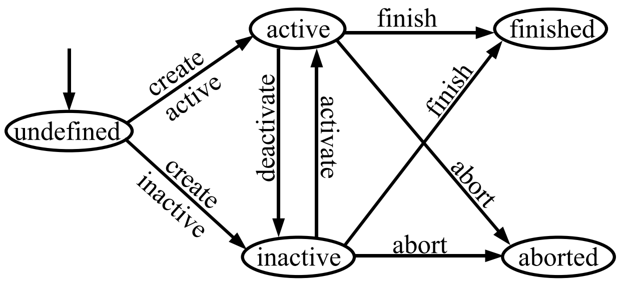


Figure 7.5: Task state transition diagram

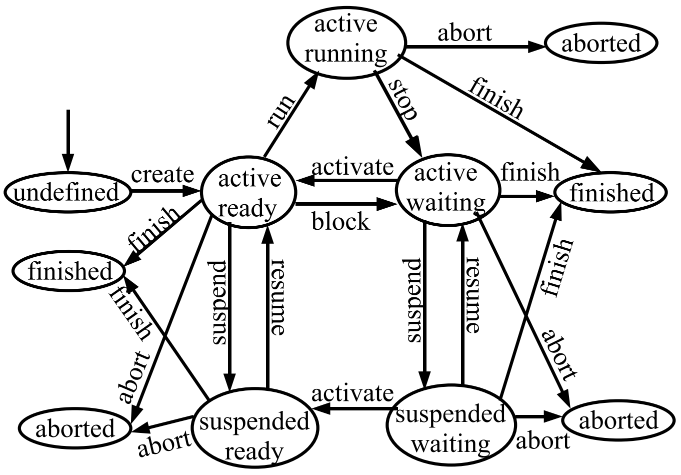


Figure 7.6: Method state transition diagram

7.5 Log Components

The simulator may write the entire event stream generated by the simulator loop and the scheduler to a log file via the log writer component. This log file can be played by a log player so that the same system behaviour can later be studied without having to run the simulator and the scheduling algorithm on the same input data again. The log player produces the same event stream as previously generated by the interaction between scheduler and simulator. This event stream is passed on to the selected visualisation components just as any stream originating from current simulation runs.

7.6 Visualisation Components

The visualisation components receive their input from the simulator or log player and display relevant data from the event streams resulting from the (live or recorded) interaction between the simulation of the behaviour of an application and the decisions made by the scheduler. The different visualisation modes focus on different aspects of the available data as we will demonstrate in this section.

7.6.1 Log View Mode

This view mode simply translates the event stream into a human-readable form. All information that can be derived from the behaviour of simulator and scheduler is accessible this way; however, the representation is obviously not very easy to understand and mostly suitable for debugging purposes.

7.6.2 Time View Mode

This view mode is an extended form of Gantt charts, displaying the state of the task set on a common time line. Information like the release and termination of instances, their activation and usage of resources and processors as well as the quality values achieved by individual nodes can be shown over a rather wide time range. On the other hand, the level of detail is comparatively small. The states of the nodes are colour-coded and additionally reflected by the width of the corresponding lines. For processors and other resources, the current utilisation is coded into the width of the line. Quality values show up above the respective node for the relevant time instants. User interaction for this view mode includes depth-first, breadth-first and manual sorting of rows, selection of the task set to display, indentation, display of static node attributes within the name tag, and zooming of the display area. Figure 7.7 shows the time view for an example simulation run.

7.6.3 Graph View Mode

The graph view mode shows the application graph in the same layout as the graph editor; during the simulation or log playing, task states, value changes and other dynamic parameters of the nodes are displayed within the nodes in addition to the pictograms used in the graph editor. Again, states are both colour-coded (the node assumes various base or frame colours) and represented by symbols to allow the state to be recognisable on monochrome media. For processors and other resources, their utilisation is shown within the node at all times. This mode gives a more

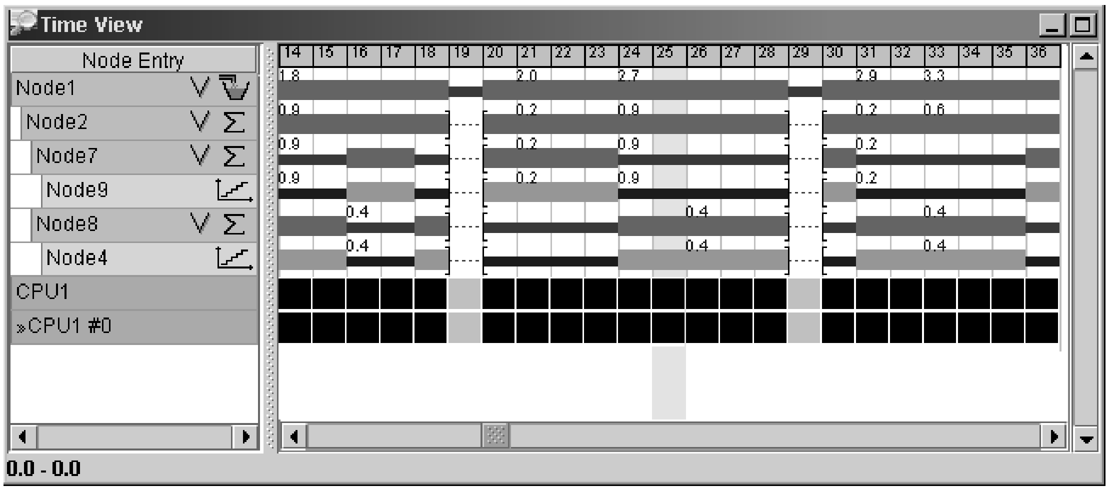


Figure 7.7: Time view mode

detailed view of the task states at a specific time, but does not allow the same big picture of the development of states over time, even though an additional slider allows to view the system state at different times in the past. Figure 7.8 is a screenshot of PaSchA's graph view mode.

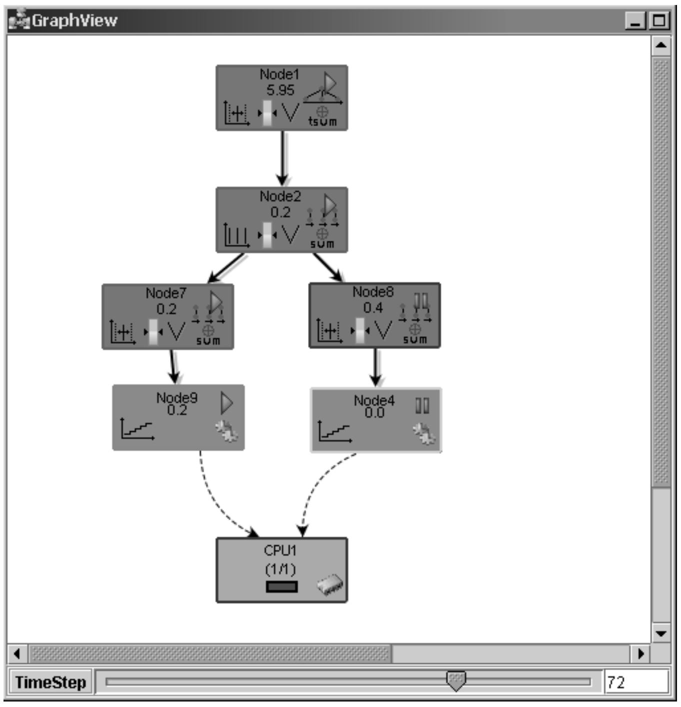
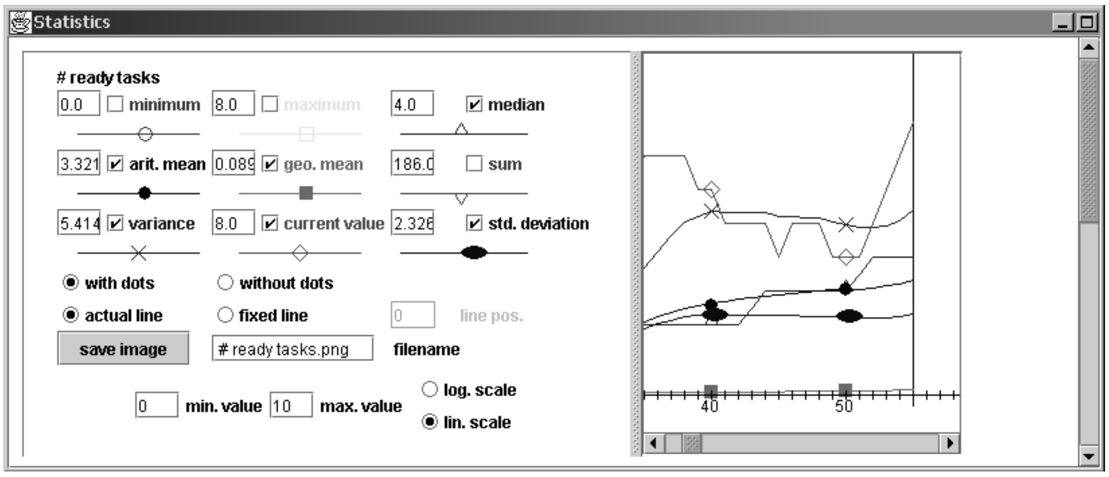


Figure 7.8: Graph view mode

7.6.4 Statistics View Mode

Finally, the statistics view mode gives the user the opportunity to derive secondary data on the application and the schedule. Among the quantities that can be included here are processor utilisation, residence time or waiting time of tasks, or the number of ready or working tasks at any time. From any individual quantity, several additional pieces of information can be recorded, e.g., arithmetic and geometric mean, median, minimum, maximum, and standard deviation. Diagrams can be plotted in either linear or logarithmic scale and exported to an image file to facilitate the inclusion into text documents. An example can be seen in figure 7.9.


Statistics

ready tasks

0.0 ☐ minimum | 8.0 ☐ maximum | 4.0 ☑ median

3.321 ☑ arit. mean | 0.089 ☑ geo. mean | 186.0 ☐ sum

5.414 ☑ variance | 8.0 ☑ current value | 2.326 ☑ std. deviation

◉ with dots ○ without dots

◉ actual line ○ fixed line 0 line pos.

save image # ready tasks.png filename

0 min. value 10 max. value ○ log. scale ◉ lin. scale

Figure 7.9: Statistics view mode

7.7 Scheduler Components

Scheduling algorithms for the PaSchA system must be implemented in Java and extend the scheduler base class provided by the system. As a minimum requirement, a scheduler class must implement three methods:

- a validity test to determine whether a given application graph is suitable for the scheduling algorithm
- an initialising method for the scheduler, which is typically the main place of action for static schedulers
- a method for execution at each point in simulator discrete time, which is used for dynamic schedulers and for dispatchers in static scheduling schemes

The scheduler base class provides some basic sanity checks and additional auxiliary functions that may simplify the implementation of scheduling algorithms. Among the optional components of a scheduler class is a scheduler-specific graphical user interface to edit the parameters in the configuration editor.

7.8 Benchmark and External Components

In order to perform automatic benchmark tests for scheduling algorithms, the appropriate application graph or generator configuration along with the necessary parameter sets are defined within Java test cases. The input data as well as the results produced by the scheduler and its interaction with the simulator are stored in a database for later evaluation. Test cases may use stored application graphs as input or generate generic loads with a graph generator. The decision in favour of specialised Java implementations for automatic benchmarking instead of a proprietary language description or a graphical tool was made due to the higher flexibility.

Apart from the support of database access, another example for an external tool that can be connected to PaSchA is the commercial linear constraint solving software CPLEX®[2]. The interface to this linear programming application is especially useful for calculating optimal solutions to many scheduling problems as a basis for performance evaluation for other algorithms.

[2]CPLEX is a registered trademark of ILOG, Inc.

Chapter 8

Experimental Results

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

Isaac Asimov

This chapter documents the results gained from simulation of generic example applications with the various scheduling algorithms emphasising different parameter settings for schedulers or certain characteristics of the application graph. We are first going to introduce some of these criteria that can be calculated from an application graph offline. After that, the performance of the schedulers introduced in earlier chapters will be evaluated according to these criteria. These experimental data were gained by simulation in the PaSchA environment (cf. chapter 7) on an Intel Pentium®4 CPU[1] with 2.0 GHz and 224 MB RAM running Java version 1.4.1 and Eclipse 3.0[2].

8.1 Application Parameters

As a first, simple example application to demonstrate a set of parameters used during the benchmark tests, consider the graph of figure 8.1 with root task T_0 and subtasks $T_1, \ldots, T_4$ in the task

[1]Intel and Pentium are registered trademarks of Intel, Inc.

[2]Eclipse is available from the Eclipse Foundation under Common Public License and Eclipse Public License agreements.

hierarchy (continuous lines), empty dependency graph and usage specifications (dashed arrows) for methods $M_1, \ldots, M_6$ running on a single processor P (not shown).

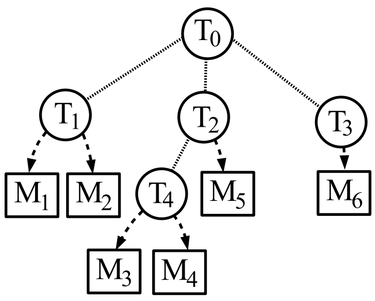


Figure 8.1: Example application

Assume the following release specifications for tasks:

task	instantiation type	additional information
T_0	sporadic	–
T_1	sporadic	–
T_2	periodic	$per_{T_2} = 5$, $j_{T_2} = 2$
T_3	aperiodic	$iat_{T_3} = 2$, $p_{T_3} = 0.5$
T_4	sporadic	–

Let the quality and utility functions be as follows:

$$\forall t \geq 0 : u_{T_0}(t) = u_{T_1}(t) = u_{T_4}(t) = 1$$

$$u_{T_2}(t) = \begin{cases} 1 & \text{if } 0 \leq t < 3 \\ 0.6 & \text{if } 3 \leq t < 5 \\ 0 & \text{if } t \geq 5 \end{cases} \qquad u_{T_3}(t) = \begin{cases} 1 & \text{if } 0 \leq t < 2 \\ 0.3 & \text{if } 3 \leq t < 3 \\ 0 & \text{if } t \geq 3 \end{cases}$$

$$q_{M_1}(n) = \begin{cases} 0 & \text{if } n < 2 \\ 0.3 & \text{if } 2 \leq n < 4 \\ 0.8 & \text{if } n \geq 4 \end{cases} \qquad q_{M_2}(n) = \begin{cases} 0 & \text{if } n < 3 \\ 0.6 & \text{if } 3 \leq n < 4 \\ 0.9 & \text{if } n \geq 4 \end{cases}$$

$$q_{M_3}(n) = \begin{cases} 0 & \text{if } n < 4 \\ 0.5 & \text{if } 4 \leq n < 5 \\ 1.0 & \text{if } n \geq 5 \end{cases} \qquad q_{M_4}(n) = \begin{cases} 0 & \text{if } n < 2 \\ 0.3 & \text{if } 2 \leq n < 4 \\ 0.7 & \text{if } n \geq 4 \end{cases}$$

$$q_{M_5}(n) = \begin{cases} 0 & \text{if } n < 3 \\ 0.8 & \text{if } 3 \leq n < 4 \\ 0.9 & \text{if } n \geq 4 \end{cases} \qquad q_{M_6}(n) = \begin{cases} 0 & \text{if } n < 3 \\ 0.5 & \text{if } 3 \leq n < 4 \\ 1.0 & \text{if } n \geq 4 \end{cases}$$

8.1.1 Method Execution Times

In quality-flexible models, the run-to-completion assumption does not apply. Therefore, there is no execution time inherent to a method. Instead, it is part of the duties of the scheduler to decide on the execution time of each method instance. However, we can rate the execution time of a method relative to a given real value; the threshold execution time is the minimum processing time needed for an instance of the method to surpass this threshold in quality:

Definition 36 (Quality threshold and threshold execution time)
For $\vartheta_q \in [0;1]$ *(the quality threshold), we define the threshold execution time* $\eta_M^{\vartheta_q}$ *of* M *as*

$$\eta_M^{\vartheta_q} := \begin{cases} \min\{n \in \mathbb{LT}_{P,I_M^k} : q_{P,M}(n) \geq \vartheta_q, P \in \mathbb{P}, k \in \mathbb{N}_0\} & \textit{if there is such an } n \\ \infty & \textit{otherwise} \end{cases}$$

For the example graph and a quality threshold of 0.8, we receive the following execution times:

$$\eta_{M_1}^{0.8} = 4 \quad \eta_{M_2}^{0.8} = 4 \quad \eta_{M_3}^{0.8} = 5 \quad \eta_{M_4}^{0.8} = \infty \quad \eta_{M_5}^{0.8} = 3 \quad \eta_{M_6}^{0.8} = 4$$

8.1.2 Task Deadlines

Similarly, in timeliness-flexible models, there is no scalar deadline for tasks. We can, however, define deadlines relative to a real value; the threshold deadline is the maximum time allowed for an instance of the task before its utility drops below this given threshold:

Definition 37 (Utility threshold and threshold deadline)
For $\vartheta_u \in [0;1]$ *(the utility threshold), we define the threshold deadline* $\delta_T^{\vartheta_u}$ *as follows:*

$$\delta_T^{\vartheta_u} := \begin{cases} \min\{t \in \mathbb{GT} : t > 0 \wedge u_T(t) \leq \vartheta_u\} & \textit{if there is such a } t \\ \infty & \textit{otherwise} \end{cases}$$

In the example application and for a utility threshold of 0.1, we have

$$\delta_{T_0}^{0.1} = \infty \quad \delta_{T_1}^{0.1} = \infty \quad \delta_{T_2}^{0.1} = 5 \quad \delta_{T_3}^{0.1} = 3 \quad \delta_{T_4}^{0.1} = \infty$$

8.1.3 Mean Interarrival Time

The number of instances of each task in any given interval of time is of course determined by the frequency of instance releases or the mean time between consecutive instance releases of the same task. In our model, only a small number of different release specifications were introduced, so that it is easy to state the mean interarrival time for a task as follows:

Definition 38 (Mean interarrival time)
The mean interarrival time for a task T is defined as:

$$\overline{iat}_T = \begin{cases} \infty & \textit{if } T \in \mathbb{T}_1 \\ per_{T'} & \textit{if } T' \in a(T) \cap \mathbb{T}_p \\ iat_{T'} + \frac{1}{p_{T'}} - 1 & \textit{if } T' \in a(T) \cap \mathbb{T}_a \end{cases}$$

In the given example, we receive for the mean interarrival times:

$$\overline{iat}_{T_0} = \infty \quad \overline{iat}_{T_1} = \infty \quad \overline{iat}_{T_2} = 5 \quad \overline{iat}_{T_3} = 2 + \frac{1}{0.5} - 1 = 3 \quad \overline{iat}_{T_4} = per_{T_2} = 5$$

8.1.4 Threshold Utilisation

For unstructured task sets under the run-to-completion assumption, utilisation is usually defined as the sum of the quotient of execution time and interarrival time for each task. The two differences in our situation are the lack of a unique execution time on the one hand and the hierarchical task structure on the other. We solve the first problem by parameterising the definition for utilisation by a quality threshold and using threshold execution times, and the second one by a recursive definition along the hierarchy of the tasks. We are primarily interested in long-term utilisation levels (i.e., in permanent overload situations). We therefore define the contribution from sporadic tasks as 0. Definitions for transient overloads should include non-zero contributions from sporadic tasks and depend also on threshold deadlines, not only on the mean interarrival times of tasks. Furthermore, we assume that a method not being able to reach the desired minimum quality (i.e., $\eta_M^{\vartheta_q} = \infty$) may use up to one full cycle of the calling task in computation time[3]. Utilisations for subtasks are summed up to yield the utilisation of the parent task. We define the (long-term) threshold utilisation of an application as follows:

Definition 39 (Threshold utilisation)
The threshold utilisation for a task T and quality threshold ϑ_q is defined as

$$U_T^{\vartheta_q} := \frac{1}{\overline{iat}_T} \cdot \sum_{M \in c(T) \cap \mathbb{M}} \min(\overline{iat}_T, \eta_M^{\vartheta_q}) + \sum_{T' \in c(T) \cap \mathbb{T}} U_{T'}^{\vartheta_q} \qquad \textit{where} \qquad \frac{1}{\infty} := 0.$$

[3]hence the minimum operator in the definition of the threshold utilisation; the definition always restricts the allocation to a task to one full cycle of the calling task, even if $\eta_M^{\vartheta_q}$ is finite.

The threshold utilisation $U_{\mathbb{T}'}^{\vartheta_q}$ for a task set $\mathbb{T}'$ with root node T_0 is defined as the threshold utilisation of the root node $U_{T_0}^{\vartheta_q}$.

In the example, the utilisation for a quality threshold of 0.8 is

$$\begin{aligned}
U_{T_1}^{0.8} &= \frac{1}{\overline{iat}_{T_1}} \cdot (\min(\overline{iat}_{T_1}, \eta_{M_1}^{0.8}) + \min(\overline{iat}_{T_1}, \eta_{M_2}^{0.8})) = \frac{1}{\infty} \cdot (4+4) = 0 \\
U_{T_4}^{0.8} &= \frac{1}{\overline{iat}_{T_4}} \cdot (\min(\overline{iat}_{T_4}, \eta_{M_3}^{0.8}) + \min(\overline{iat}_{T_4}, \eta_{M_4}^{0.8})) = \frac{1}{5} \cdot (5+5) = 2 \\
U_{T_2}^{0.8} &= \frac{1}{\overline{iat}_{T_2}} \cdot \min(\overline{iat}_{T_2}, \eta_{M_5}^{0.8}) + U_{T_4}^{0.8} = \frac{1}{5} \cdot 3 + 2 = 2.6 \\
U_{T_3}^{0.8} &= \frac{1}{\overline{iat}_{T_3}} \cdot \min(\overline{iat}_{T_3}, \eta_{M_6}^{0.8}) = \frac{1}{3} \cdot 3 = 1 \\
U_{T_0}^{0.8} &= U_{T_1}^{0.8} + U_{T_2}^{0.8} + U_{T_3}^{0.8} = 0 + 2.6 + 1 = 3.6
\end{aligned}$$

8.1.5 Standard Deviation of Task Release Times

The instance density of non-sporadic tasks equals their release frequency. However, for sporadic tasks higher in the hierarchy, the instance density is the average rate of release of any of its direct or indirect children. This is a first step to compute an aggregate value for the standard deviation of task release times from their estimate along the task hierarchy.

The instance density of a task (sub)tree is defined as

Definition 40 (Instance density)
The instance density of a task T is defined as:

$$id_T := \max\left(\frac{1}{\overline{iat}_T}, \sum_{T' \in c(T) \cap \mathbb{T}} id_{T'}\right)$$

The standard deviation of the release time of a task is computed locally for each task regardless of its position within the hierarchy.

Definition 41 (Standard deviation of release time)
The standard deviation of the release time of task T is

$$std_T := \begin{cases} 0 & \textit{if } T \in \mathbb{T}_s \\ \frac{j_T}{2} & \textit{if } T \in \mathbb{T}_p \\ \frac{\sqrt{1-p_T}}{p_T} & \textit{if } T \in \mathbb{T}_a \end{cases}$$

The definitions of the weighted average standard deviation of the release times for child tasks of T and the aggregate standard deviation of release times depend on each other. From the standard deviations of the child node release times, aggregate values are computed as given below. If the instance densities among the children of a task are non-zero, they are used as weights to calculate the weighted sum of the standard deviations of the child node release times. Otherwise, the arithmetic mean is computed instead.

Definition 42 (Weighted average standard deviation of release times for children)

The weighted average standard deviation of the release times for the child nodes of task T is

$$stcd_T := \begin{cases} 0 & \textit{if } \mathbb{T} \cap c(T) = \emptyset \\ \frac{1}{|\mathbb{T}\cap c(T)|} \cdot \sum_{T' \in \mathbb{T}\cap c(T)} std^*_{T'} & \textit{if } \forall T' \in \mathbb{T} \cap c(T) : id_{T'} = 0 \\ \frac{1}{\sum_{T' \in \mathbb{T}\cap c(T)} id_{T'}} \cdot \sum_{T' \in \mathbb{T}\cap c(T)} id_{T'} \cdot std^*_{T'} & \textit{otherwise} \end{cases}$$

Definition 43 (Aggregated standard deviation of release times)
The aggregated standard deviation of release times for a tree with root node T is

$$std^*_T := \max(std_T, stcd_T)$$

The aggregated standard deviation of release times for the root task $std^*_{T_0}$ is a measure for the degree of nondeterminism in the application graph. If all release times are deterministic, we know that $std^*_{T_0} = 0$. With decreasing reliability of the release time estimates, $std^*_{T_0}$ increases. The values calculated at each level in the hierarchy are weighted by the frequency of the respective child tasks or subtrees as far as possible.

The example application has following values for instance densities and standard deviations of release times:

T	id_T	std_T	$stcd_T$	std^*_T
T_1	0	0	0	0
T_4	$\frac{1}{5}$	0	0	0
T_2	$\frac{1}{5}$	1	$\frac{\frac{1}{5}\cdot 0}{\frac{1}{5}} = 0$	1
T_3	$\frac{1}{3}$	$\frac{\sqrt{1-0.5}}{0.5} = 1.42$	0	1.42
T_0	$\frac{8}{15}$	0	$\frac{0\cdot 0+\frac{1}{5}\cdot 1+\frac{1}{3}\cdot 1.42}{0+\frac{1}{5}+\frac{1}{3}} = 1.26$	1.26

8.2 Basic Performance Tests for Scheduling Algorithms

In the first series of benchmark tests we primarily rate the correlation of execution time of the scheduler and the value of the resulting schedules for task sets with deterministic release times. The tests in this section do not take into account the cost of scheduling as additional load on the processor(s). In other words, these performance test are aimed at the simple model without feedback mechanism we introduced in the first four chapters of this thesis.

Whereas interrupting the search and optimisation algorithms at an arbitrary time is usually possible for unstructured task sets or flat hierarchy graphs, the hierarchical structure and the resulting recursive calling order of the algorithm for individual nodes renders the interruptible anytime model unfeasible for general application graphs. Therefore, the execution time of the schedules is determined via the choice of appropriate parameters. In terms of the classification scheme of section 1.3.2, the scheduling task is of the parametrisation type of flexibility. The parametrisation schemes of scheduling algorithms are not very fine-granular; this explains the irregular distribution of sampling points in the time-value diagrams of this section. Of course, the application tasks are of the external termination type (figure 1.3).

Parameters common to all our scheduling algorithms are the quality and utility threshold used to limit the search space. They should be chosen as a reasonable compromise between closeness to the original specification (high quality threshold and low utility threshold) and efficiency of scheduling. Lower quality thresholds prevent minor quality changes in the initial phase of methods from being taken into account during the optimisation process; higher utility thresholds allow the scheduler to remove tasks unlikely to contribute to the overall performance at an early stage. Both of these effects reduce the size of the search space.

In order to compare different algorithms and parameter settings for them, we apply them to the same application with the following characteristics:

number of tasks	10
task instances within 100 simulation steps	80
average number of quality levels per method	6
average number of utility levels per task	6
threshold utilisation $U_{T_0}^{0.1}$	0.98
threshold utilisation $U_{T_0}^{0.5}$	2.87
threshold utilisation $U_{T_0}^{0.8}$	3.96
aggregate standard deviation of release times $std_{T_0}^{*}$	0

The hierarchy graph is flat, i.e., it consists of a root node, one layer of child tasks and the underlying method layer. The main factors influencing the complexity of the scheduling problem are the number of task instances within the scheduling window (depending on the number of tasks and their mean interarrival time), the number of utility levels per task and the utilisation for given quality thresholds. The impact of the aggregate standard deviation of release times depends on the type of scheduling algorithm and the strategy for dealing with unexpected application behaviour.

8.2.1 Simulated Annealing Algorithm

As the parameters for the simulated annealing scheduler (initial temperature, minimum temperature, cool-down factor, number of search steps in temperature equilibrium) are quite complex in their influence on the performance of the scheduling algorithm, they were set automatically according to a given $ratio$ of search steps and the size of the search space: For a given search space size $ss(\mathbb{T}', \mathbb{J})$ and arbitrary temperatures

$$Temp_{start} > Temp_{end} > 0,$$

we define

$$\#rep := \left\lceil \sqrt{ratio \cdot ss(\mathbb{T}', \mathbb{J})} \right\rceil$$

$$cF := \left(\frac{Temp_{end}}{Temp_{start}} \right)^{\frac{1}{\#rep}}$$

The search ratio is the primary source of altering the computation time of the algorithm.

Further parameters are the size of the scheduling window and the minimum delay between consecutive scheduling phases. Rescheduling takes place no earlier than the minimum delay from the previous scheduling phase and obviously no later than the end of the partial schedule. We will present test results for scheduling with different window sizes and generally adapt the minimum delay accordingly, such that the window size is a fixed multiple of the minimum delay for different runs of a test series.

Next, parameters exist for the size of the local cache of solutions for each task node, for the computation or estimation of the search space size, and the translation of the search space size into probabilities for the selection of intervals during the search (compare section 3.1.2.4). Finally, minimum and maximum numbers of search steps and the distance in processing time units covered by a single search step can be set manually. We will not, however, present experimental results for these parameters, as they either do not seem to influence performance significantly or a very limited range of reasonable values for them is determined by the other parameters and the given application graph.

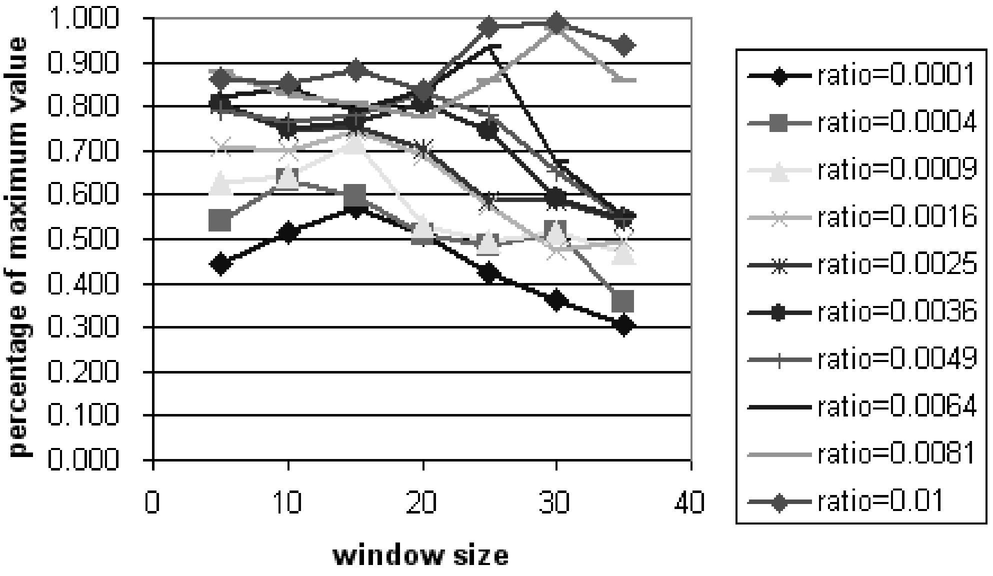


Figure 8.2: Window size - value diagram for simulated annealing (deterministic release times)

Figures 8.2 and 8.3 show the original relationship between parameter settings for the simulated annealing scheduler on the one hand and the schedule value and the execution time on the other.

The parameter-value diagram in figure 8.2 gives evidence of the fact that a larger scheduling window results in an increased knowledge of the future set of active tasks and can thus improve the achievable value for the schedules. It appears that the profiles for different settings of the search ratio have different optimal window sizes (*ws*); these are generally smaller for lower search ratios. The scheduling effort increases with both search ratio and window size, both of these parameters directly influencing the number of search steps to be covered by the optimisation algorithm (figure 8.3). However, as our test conditions are not ideal due to the limited time resolution and interferences by the operating system and competing applications, the monitoring of the execution time constitutes only an approximation of the real values.

Instead of working with the original diagrams for execution time and schedule value in relation to scheduler parameters, we derive diagrams on the direct correlation of time and value as in figure 8.4 in the remainder of this section. The performance of the scheduler rises with computation time and reaches a plateau for all window sizes; for smaller window sizes, the slope of the profile is steeper at the beginning, but the maximum value is higher than for bigger scheduling windows.

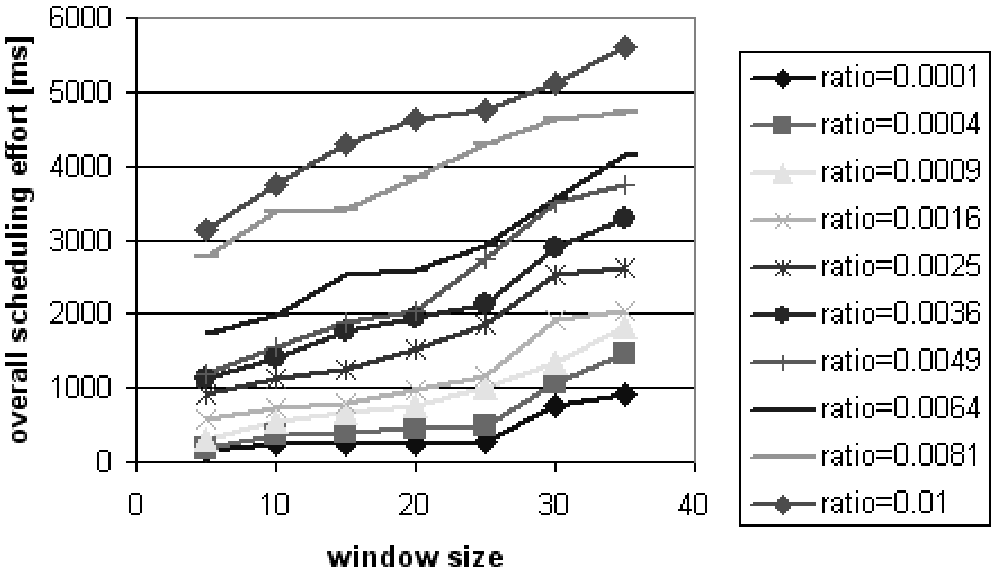


Figure 8.3: Window size - effort diagram for simulated annealing (deterministic release times)

8.2.2 Tabu Search Algorithm

The parameters of the tabu search algorithm are the length of the tabu list, the number of diversification steps for one invocation and the number of normal search steps before triggering a diversification step. We use the window size (*ws*) and the length of the tabu list to generate parameter / value and parameter / time diagrams analogous to figures 8.2 and 8.3; from these we can derive the performance profiles of figure 8.5.

The performance profiles show that the value of the schedules rises with increasing computational effort, and that higher values are achieved faster for shorter scheduling windows, i.e., smaller problem sizes. However, the performance of the scheduler is considerably and consistently lower than that of the simulated annealing alternative. Our conclusion is that the set of diversification steps chosen is too simple. As stated in the literature, tabu search is a metaheuristic whose performance depends largely on the one of the original heuristics it guides. One promising possibility to investigate could be to use tabu search as a metaheuristic to guide the simulated annealing algorithm. In other words, our simulated annealing algorithm could be improved by adding a tabu list for local search steps and allowing diversification, i.e., restarting simulated annealing search from points outside the current neighbourhood.

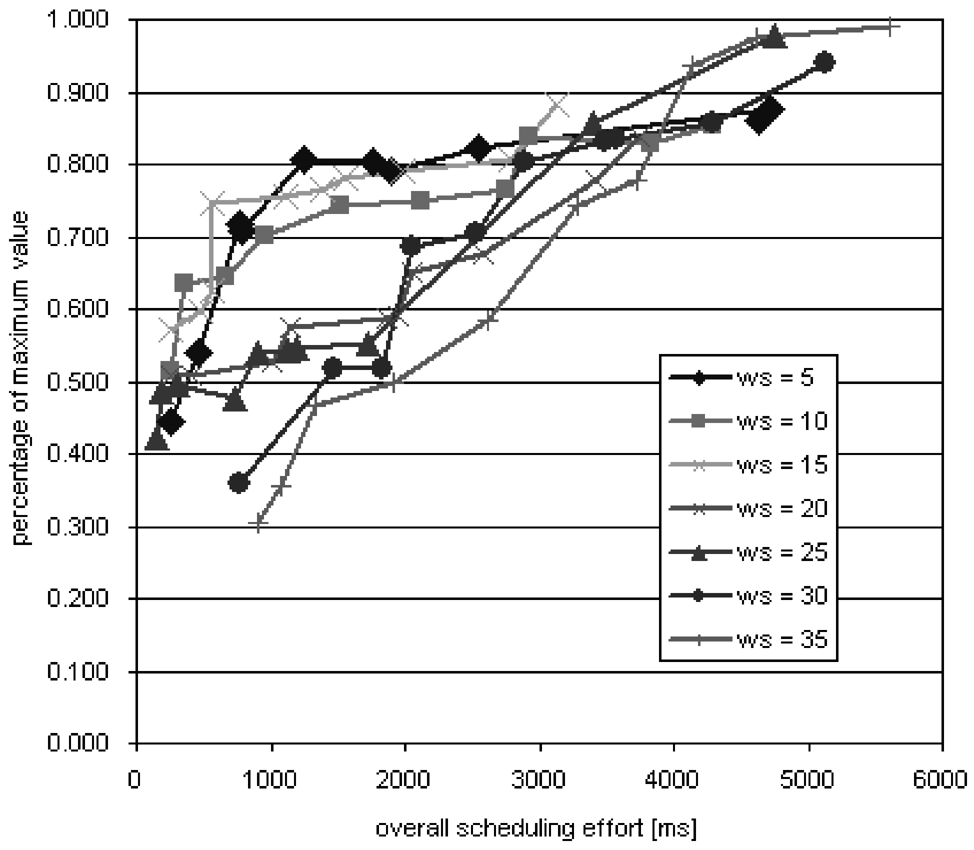


Figure 8.4: Effort / value diagram (simulated annealing, deterministic release times)

8.2.3 Lagrange Multiplier Algorithm

The parameters for the scheduler based on Lagrange multipliers are the approximation scheme for the discrete original functions (quadratic functions, quadratic splines, or B-splines) and the size of the scheduling window. The execution time of the algorithm can be influenced via the granularity of the functions, i.e., the number of defining points for interpolation. However, the scalability of the algorithm by these means is limited (by the generally small numbers of quality levels for methods and utility levels for tasks) and dominated by the window size and the corresponding size of the search space.

The performance of the algorithm depends primarily on the window size and the homogeneity of the task release times (especially the period lengths for periodic tasks) with this window size. The performance diagram of figure 8.6 shows that execution time is primarily determined by the window size and has no detectable correlation with the value of the schedules. The algorithm scales very badly with both the input parameters and the execution time, such that the value is

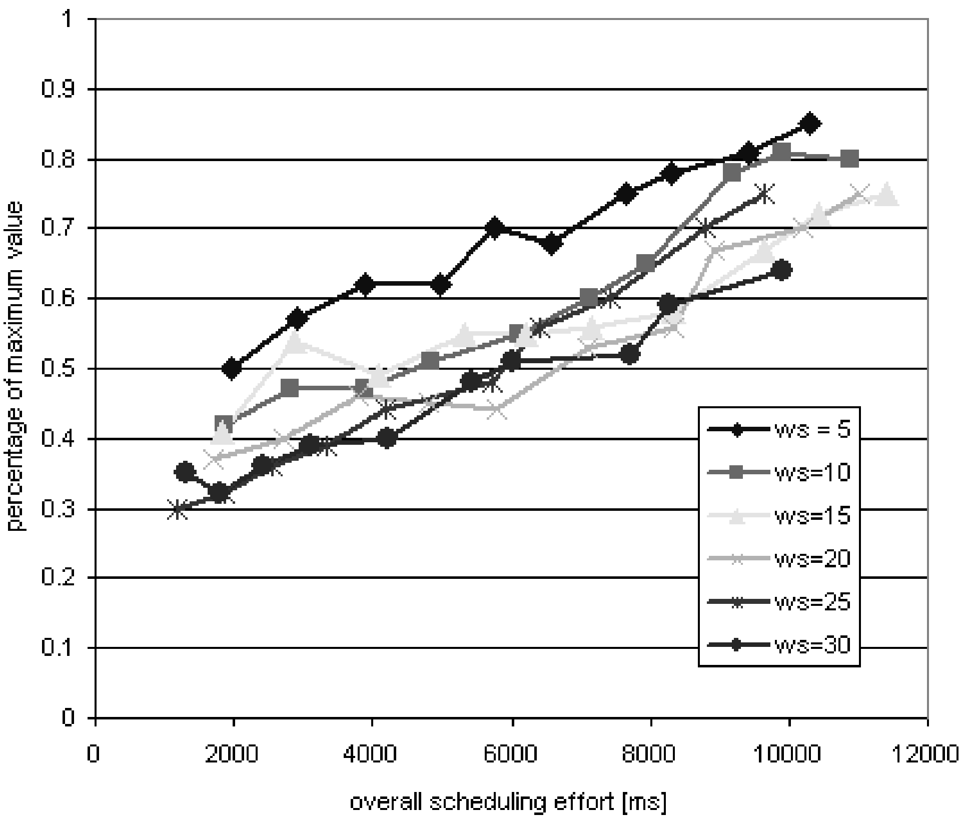


Figure 8.5: Effort / value diagram (tabu search, deterministic release times)

approximately constant for a specific setting of the window size.

8.2.4 Decision-Theoretic Algorithm

The parameters for the policy iteration algorithm of the decision-theoretic scheduler are primarily the discounting factor (*df*) and the size of the state envelope, where the envelope size determines the execution time of the algorithm.

The performance profiles of figure 8.7 show that the maximum values that can be reached with small discounting factors are quite low, because the scheduler has hardly any knowledge of the future behaviour of the application. This drawback of the decision-theoretic scheduler can be counteracted by higher values for the discounting factor. However, the processing time to reach reasonably good results is unpleasantly high, the reason being the overhead of calculating state sets despite the much simpler representation of task instances with deterministic release times. What the performance profile also shows is that the scheduler requires a relatively long

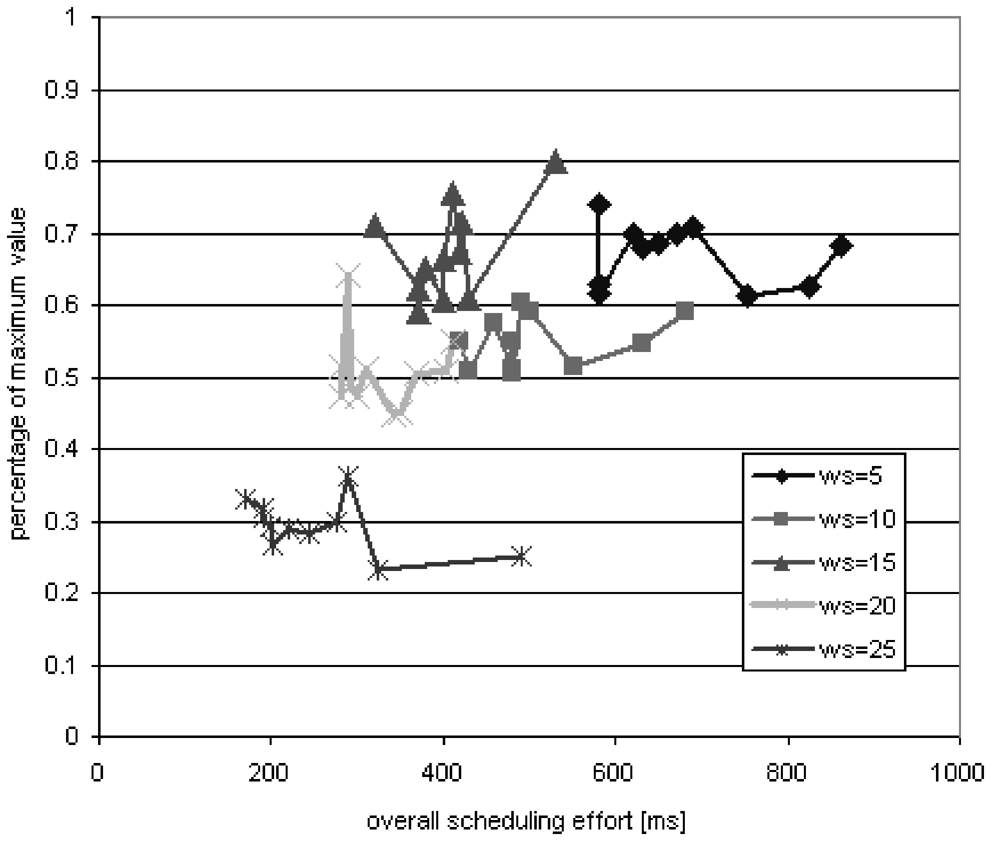


Figure 8.6: Effort / value diagram (Lagrange multipliers, deterministic release times)

processing time before producing first results; the primary application area are task sets with less predictable release times, as we will see later.

8.3 Performance Tests for Task Sets with Nondeterministic Release Times

Now we investigate the influence of nondeterminism in the release times of tasks.

In the previous section, the tabu search and Lagrange multiplier approaches appeared to be little promising. Therefore, we concentrate on the simulated annealing and decision-theoretic algorithms in the following.

For our experiments we first use a set of applications consisting of flat hierarchies of tasks with the characteristics of table 8.1.

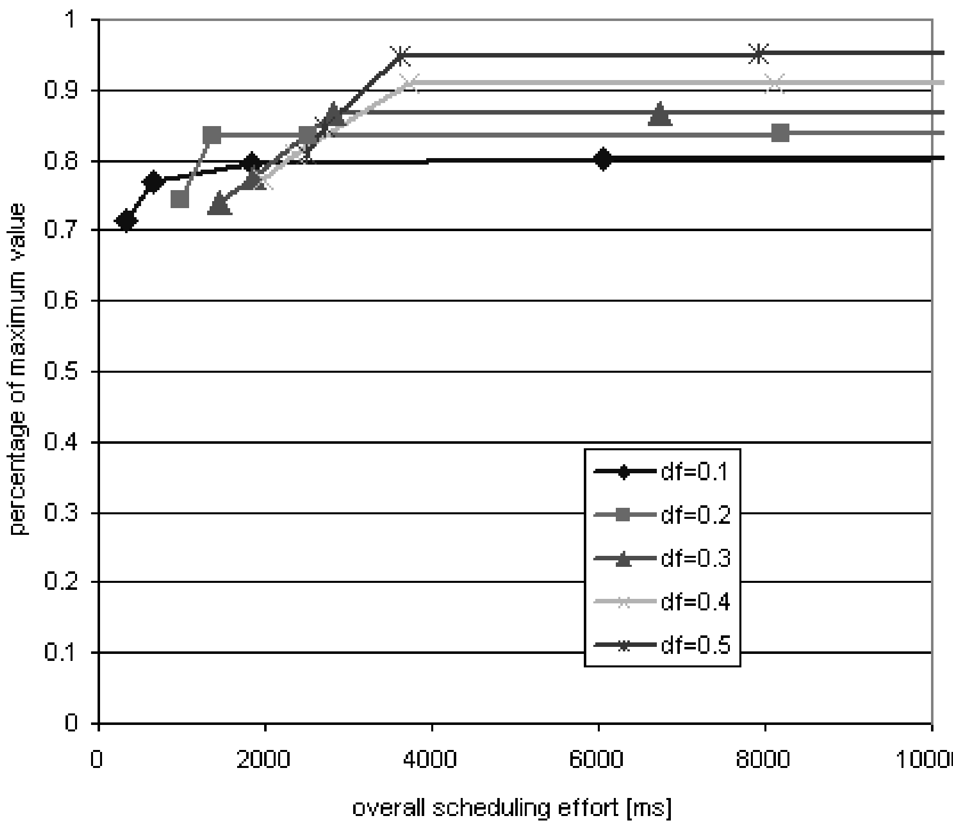


Figure 8.7: Effort / value diagram (Markov decision processes, deterministic release times)

task set $\mathbb{T}$	$std^*_{T_0}$	$U^{0.1}_{T_0}$	$U^{0.3}_{T_0}$	$U^{0.6}_{T_0}$	$U^{0.9}_{T_0}$	$\max_{T\in\mathbb{T}} \delta^{0.2}_T$
$\mathbb{T}_1$	0.00000	0.98741	2.46423	3.41979	4.09370	10
$\mathbb{T}_2$	0.41044	0.97807	2.44473	3.39069	4.06217	9
$\mathbb{T}_3$	0.79595	0.95944	2.40746	3.33390	3.99484	10
$\mathbb{T}_4$	1.72208	0.91164	2.31875	3.19334	3.80954	10
$\mathbb{T}_5$	3.14647	0.85090	2.13014	2.93888	3.54606	8
$\mathbb{T}_6$	5.37465	0.75421	1.92909	2.65997	3.16834	10
$\mathbb{T}_7$	9.48683	1.80000	4.80000	6.40000	7.80000	9

Table 8.1: Characteristics of task sets with tight deadlines

We rate these task sets as having tight deadlines with a maximum being lower than or equal to 10 for utility threshold 0.2.

Modifying the utility functions for these tasks, we can increase the threshold deadlines without changing the utilization values[4]. This way, we receive sets of tasks with lenient deadlines, the minimum being greater than or equal to 20 for utility threshold 0.2 (table 8.2).

task set $\mathbb{T}$	$std^*_{T_0}$	$U^{0.1}_{T_0}$	$U^{0.3}_{T_0}$	$U^{0.6}_{T_0}$	$U^{0.9}_{T_0}$	$\min_{T\in\mathbb{T}} \delta^{0.2}_T$
$\mathbb{T}_1$	0.00000	0.98741	2.46423	3.41979	4.09370	25
$\mathbb{T}_2$	0.41044	0.97807	2.44473	3.39069	4.06217	22
$\mathbb{T}_3$	0.79595	0.95944	2.40746	3.33390	3.99484	20
$\mathbb{T}_4$	1.72208	0.91164	2.31875	3.19334	3.80954	21
$\mathbb{T}_5$	3.14647	0.85090	2.13014	2.93888	3.54606	21
$\mathbb{T}_6$	5.37465	0.75421	1.92909	2.65997	3.16834	21
$\mathbb{T}_7$	9.48683	1.80000	4.80000	6.40000	7.80000	24

Table 8.2: Characteristics of task sets with lenient deadlines

8.3.1 Simulated Annealing Algorithm

A test series for the simulated annealing scheduler working on the sets of tasks with tight deadlines and an overall scheduling effort of approximately 700ms for each simulation run resulted in the diagram of figure 8.8.

The performance of the scheduler decreases rapidly with rising aggregate standard deviation of release times. The best choice of window size seems to be a very small value matching the quick decisions that have to be made with increasingly unreliable estimates of task release times.

The same algorithm was applied to the modified task sets for an overall scheduling time of approximately 2000ms; the longer scheduling time seems reasonable if the task deadlines are less tight, even though we do not consider the cost of scheduling explicitly in this section.

For the task sets with lenient deadlines, a higher window size is preferable, because interesting increases in quality for a task may take place even at a comparatively long time after its release, unlike in the previous case (see figure 8.9). Furthermore, the performance of the algorithm at higher levels of nondeterminism is better than in the previous case. Intuitively, inaccurate estimates of release times can be tolerated more easily if there is a longer time available to deal with the deteriorating effects.

[4]These are defined via quality functions only.

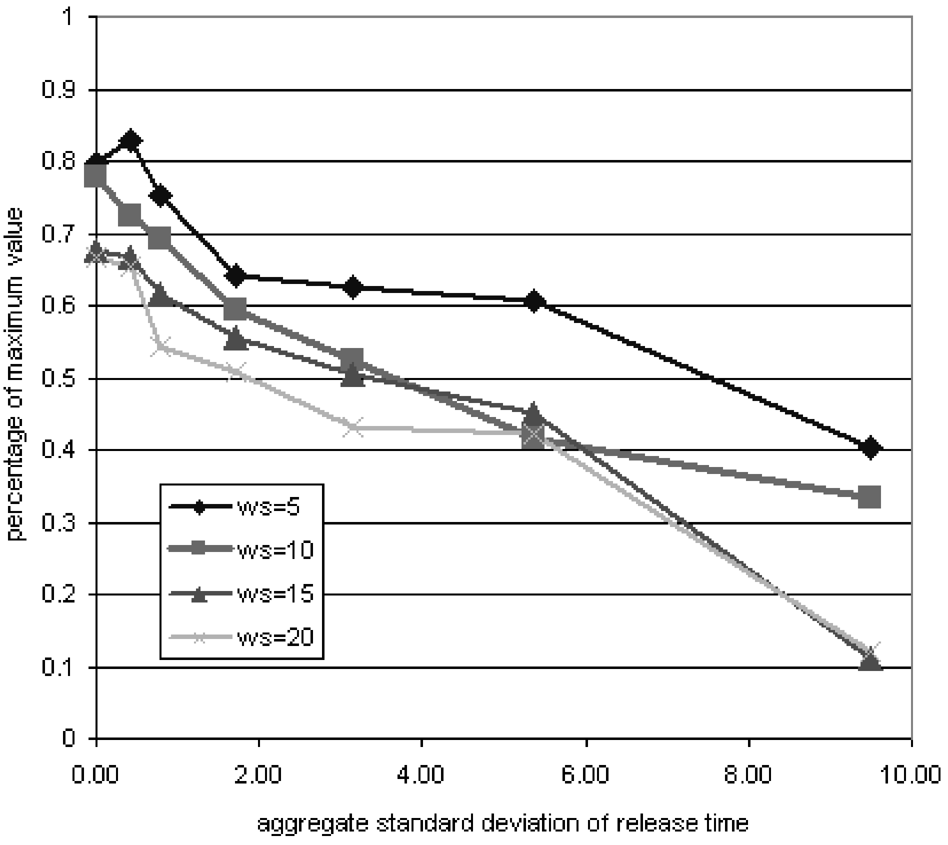


Figure 8.8: Nondeterminism / value diagram (simulated annealing, tight deadlines)

8.3.2 Decision-Theoretic Algorithm

The decision-theoretic scheduler in general requires long computation times to yield initial results, but the generation of alternative solutions can be beneficial if the release times of task instances are not very well predictable. The same task sets as above were scheduled by the decision-theoretic approach with an overall scheduling time of approximately 5000ms, resulting in the performance profiles of figure 8.10.

Of course, the performance of the scheduler decreases with rising aggregate standard deviation of release times, because the frequency of state transition steps ending in states outside the current state envelope increases. However, the performance degradation appears to be less dramatic than for the simulated annealing alternative with rescheduling. Note, though, that the computation time of the decision-theoretic scheduler is much higher in this example. Comparing the individual performance profiles, there seems to be an optimal value for the discount factor,

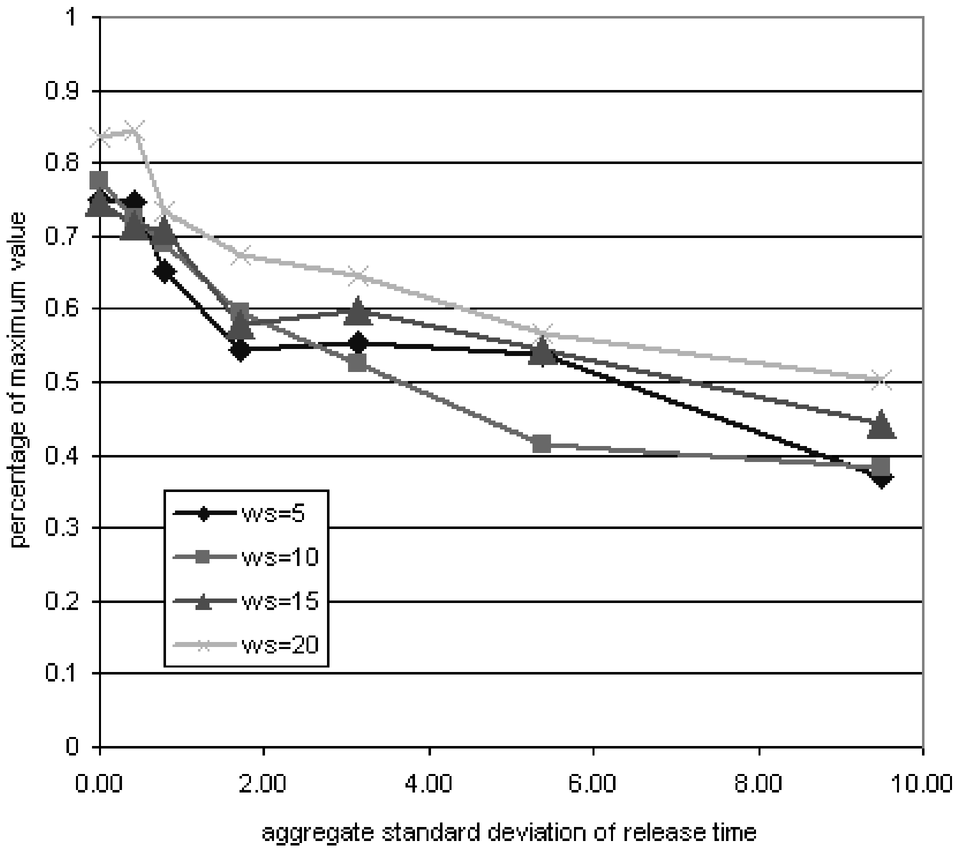


Figure 8.9: Nondeterminism / value diagram (simulated annealing, lenient deadlines)

which we assume to be related to the influence of the time intervals most interesting to the tasks of this application in the evaluation of the objective function. With the maximum threshold deadline of 10, assume the interesting quality and utility changes for a task are centred around time 5 after its release. If we want to match this time with the one when the influence of the task is equal to 0.5, we can deduce that we have to choose a discounting factor of $(0.5)^{1/5} = 0.87$.

If the second set of applications is chosen for evaluation with a scheduling time of approximately 5000ms, the profiles of figure 8.11 are the results. The main difference is the order of performance profiles for different discounting factors. With deadlines being later in these applications, a higher discounting factor is preferable, giving more emphasis to later scheduling decisions.

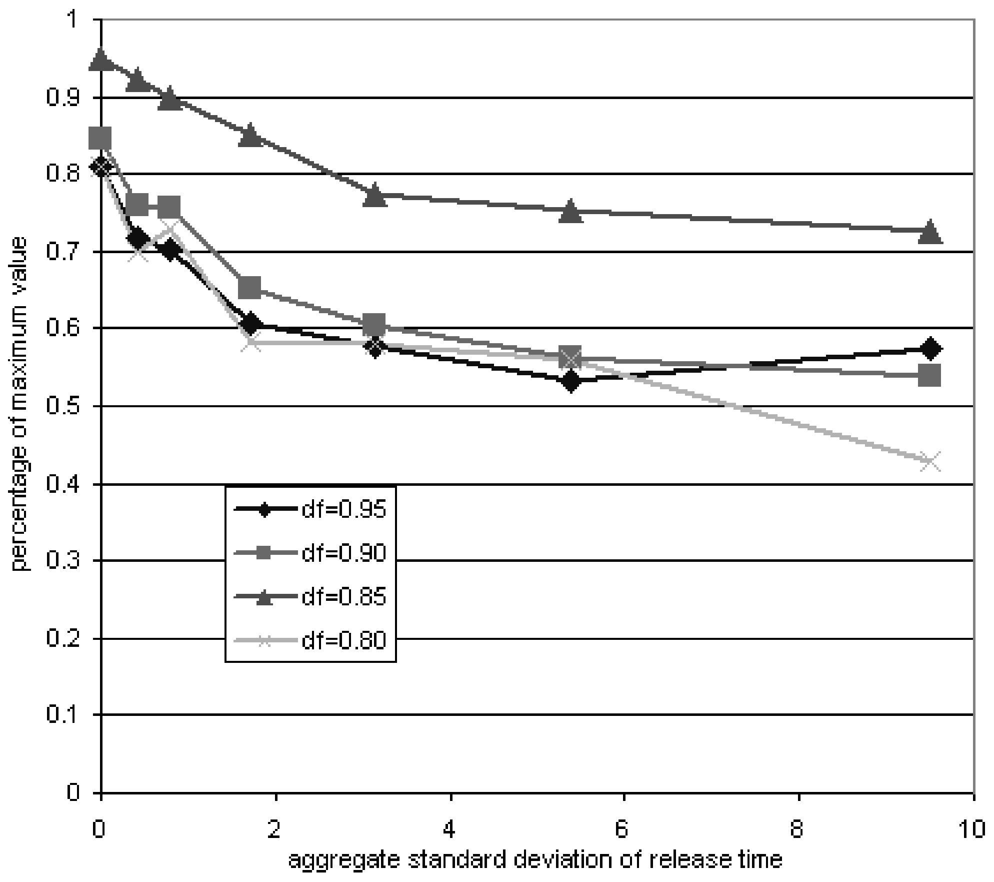


Figure 8.10: Nondeterminism / value diagram (decision-theoretic, tight deadlines)

8.3.3 Comparison of Schedulers

Comparing the simulated annealing and decision-theoretic schedulers for sets of 10 tasks similar to the ones above, but including both tasks with short and long threshold deadlines and a scheduling time of approximately 1000ms, we receive the performance profiles of figure 8.12. The main contribution of this experiment was the finding that the decision-theoretic scheduler can actually outperform the rescheduling scheme with simulated annealing at higher levels of nondeterminism (i.e., bigger deviations of the release times from their estimates).

8.4 Performance Tests for Different Utilisation Levels

In this section, we are going to present benchmark tests on task sets with different utilisation levels for the simulated annealing and decision-theoretic scheduler. Rising overload of the proces-

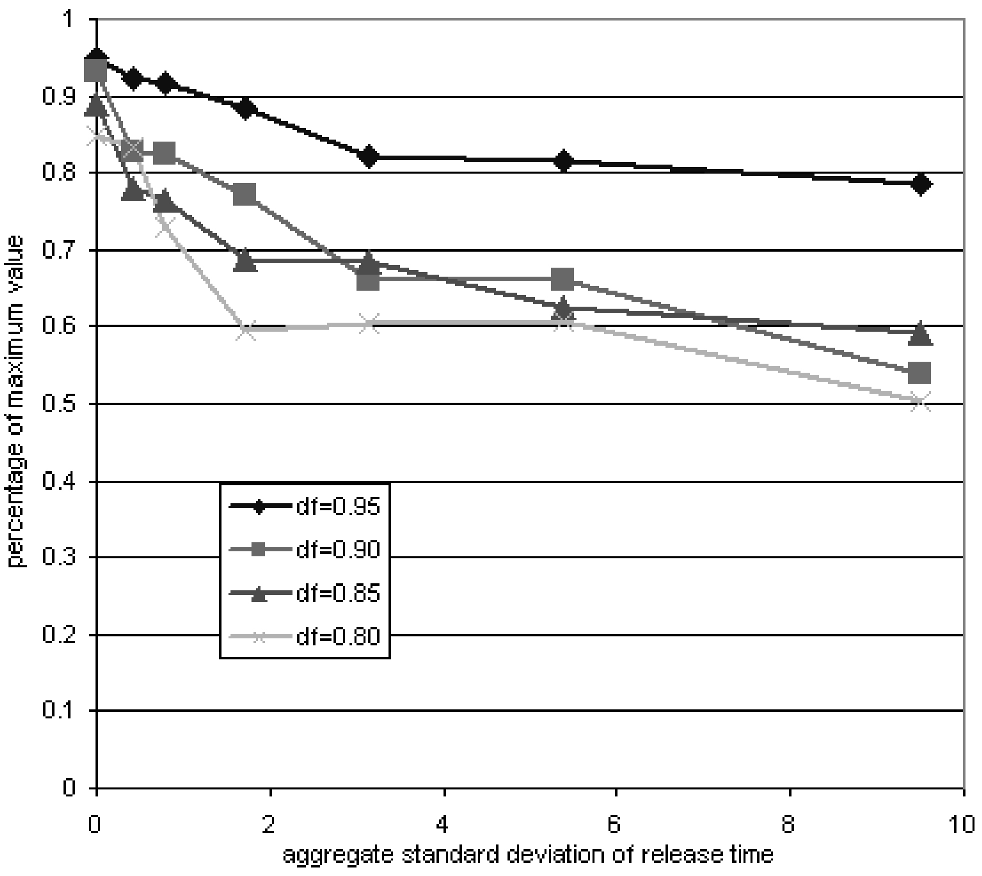


Figure 8.11: Nondeterminism / value diagram (decision-theoretic, lenient deadlines)

sor(s) means the scheduling problem becomes more complex, so that a performance degradation can be expected.

8.4.1 Deterministic Release Times

The task sets we are going to use first have deterministic release times and the utilisation specifications of table 8.3.

8.4.1.1 Simulated Annealing Algorithm

The simulated annealing algorithm degrades gracefully under increasing overload, as the performance profiles of figure 8.13 demonstrate.

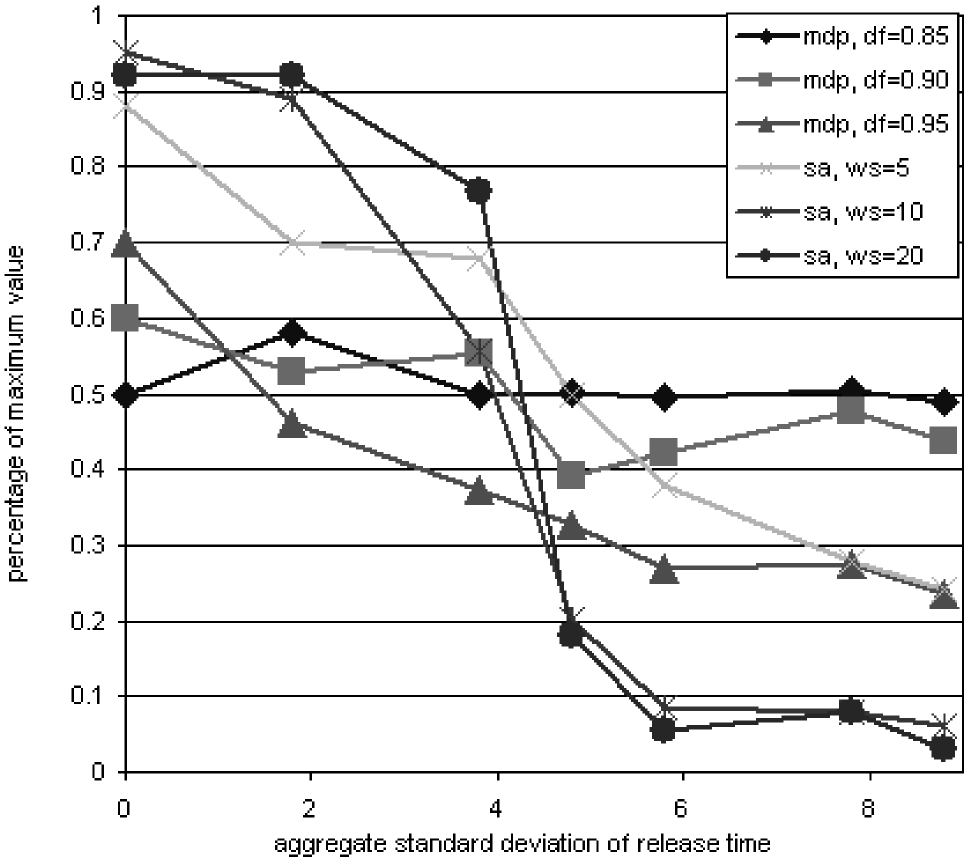


Figure 8.12: Nondeterminism / value diagram (decision-theoretic and simulated annealing)

task set $\mathbb{T}$	$std^*_{T_0}$	$U^{0.1}_{T_0}$	$U^{0.3}_{T_0}$	$U^{0.6}_{T_0}$	$U^{0.9}_{T_0}$
$\mathbb{T}_1$	0	0.30054	0.81970	1.11281	1.34732
$\mathbb{T}_2$	0	0.44781	1.21274	1.62199	2.01347
$\mathbb{T}_3$	0	0.67996	1.74952	2.32516	2.88144
$\mathbb{T}_4$	0	1.10524	2.62990	3.55418	4.42521
$\mathbb{T}_5$	0	1.79931	4.65933	6.47665	7.66358

Table 8.3: Characteristics of task sets with deterministic release times

The tasks have mostly large slack times (the time difference between service requirements and deadline for a given utility threshold), such that comparatively large settings for the window size yield the best results. Note that the performance profile for the smallest window size is quite low, but it does not seem to be influenced very much by increasing loads. As it can make deci-

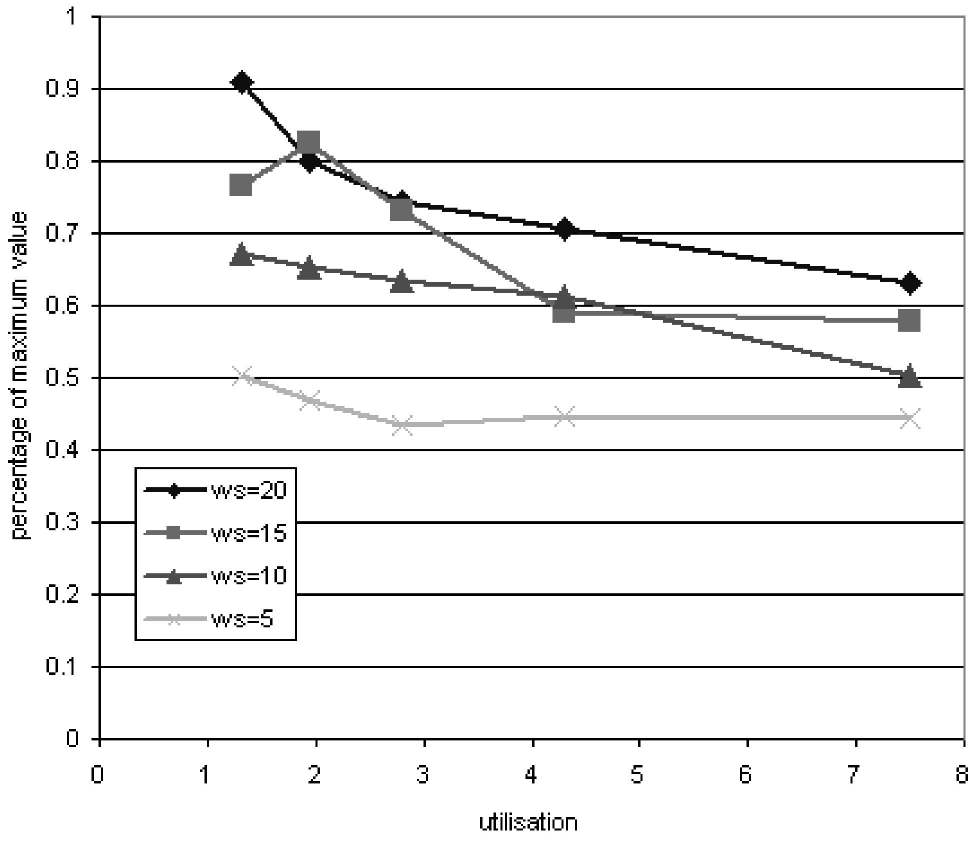


Figure 8.13: Utilisation - value diagram (simulated annealing, deterministic release times)

sions very quickly, it is obviously least affected by the rising problem size and hence increasing computational effort which makes finding optimal schedules in overload situations difficult.

8.4.1.2 Decision-Theoretic Algorithm

The decision-theoretic scheduler also degrades gracefully in overload, albeit at a lower level than the simulated annealing scheduler. In the example application with the performance profiles of figure 8.14, the highest values are achieved with a discounting factor between 0.85 and 0.9.

8.4.2 Nondeterministic Release Times

A second series of benchmark tests is performed on modified task sets with non-zero standard deviations of the task release times. As changing the release jitter of periodic tasks does not influence the mean interarrival time and hence the utilisation, it is possible to compare performance profiles gained this way directly to the ones for task sets with deterministic release times. The

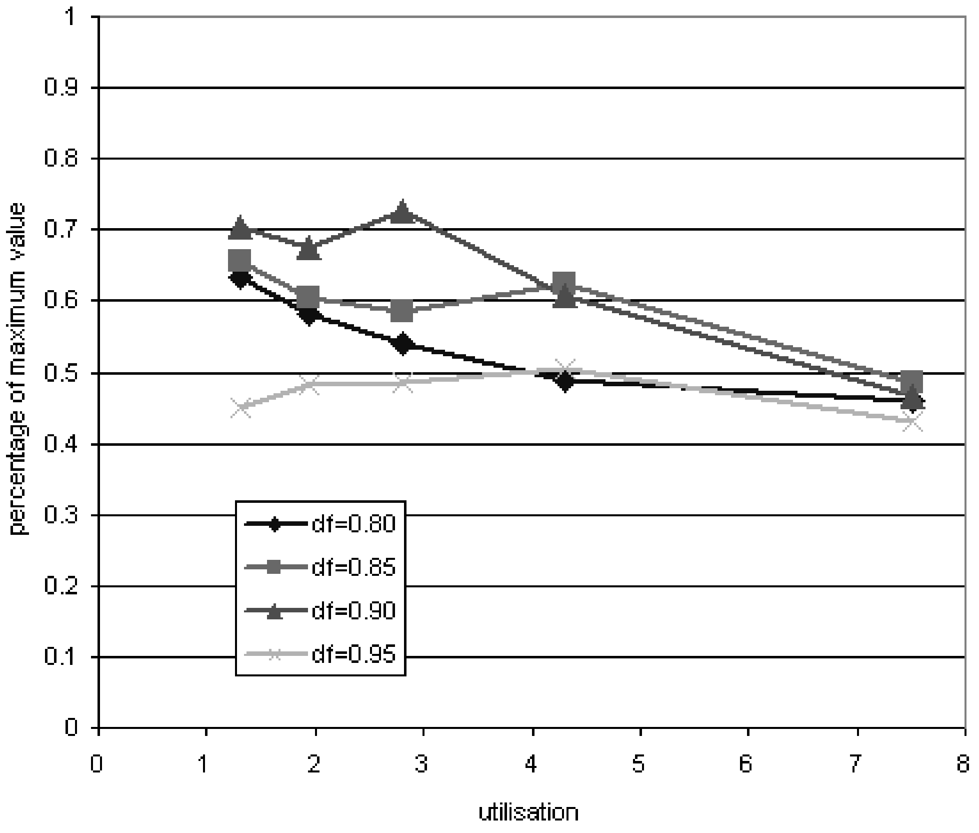


Figure 8.14: Utilisation - value diagram (decision-theoretic, deterministic release times)

utilisation specifications for the modified task sets equals the one of table 8.3, but the aggregate standard deviation of release times, $std^*_{T_0}$, ranges from 6.30 to 7.87 for the task sets.

8.4.2.1 Simulated Annealing Algorithm

Nondeterministic release times do not seem to have an additionally deteriorating effect on increasing load; even though the performance profiles of figure 8.15 for the task sets with non-zero standard deviations for the release times are lower than the ones gained for deterministic release times, the performance of the algorithm still degrades gracefully in overload.

8.4.2.2 Decision-Theoretic Algorithm

The decision-theoretic approach outperforms the simulated annealing scheduler for sets of tasks with badly predictable release times also in case of high load, as the performance profiles of 8.16 demonstrate.

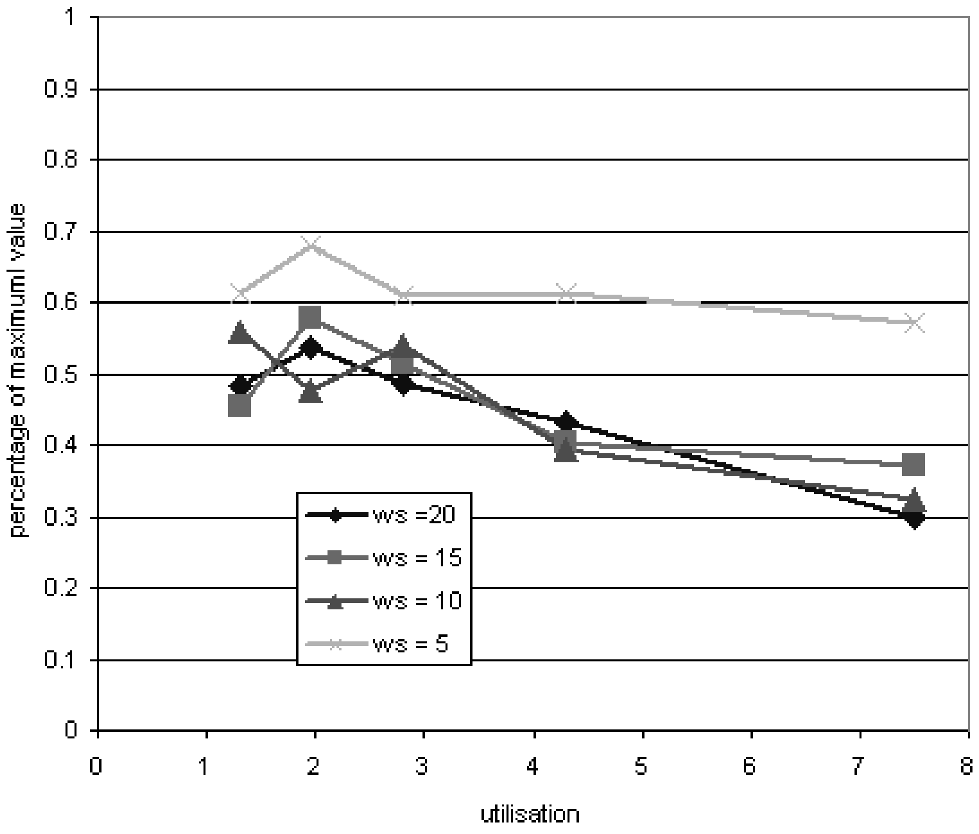


Figure 8.15: Utilisation - value diagram (simulated annealing, nondeterministic release times)

8.5 Performance Tests for Feedback Mechanism

Up to now, performance tests were made under the simplifying assumption that scheduling costs are small enough to be neglected. However, for complex algorithms used within dynamic schedulers executed on the same processor(s) as the application tasks, this assumption is not realistic. A first step to take into account the cost of scheduling is a fixed reservation of processing time for the scheduler within any window. However, this method is not very flexible, as the appropriate percentage of processor reservation for the scheduling algorithm cannot easily be determined before runtime, and the requirements may even change over time.

The approach taken in chapter 5 of this thesis was to make flexible reservations of processing time for the scheduler and use a PID controller to adapt this scheduling allowance to the current parameters of the application. Two series of tests will be described in this section. One demonstrates the stability of the controller, and the other shows that the flexible allocation of processing time to a scheduler outperforms a fixed reservation for the scheduler.

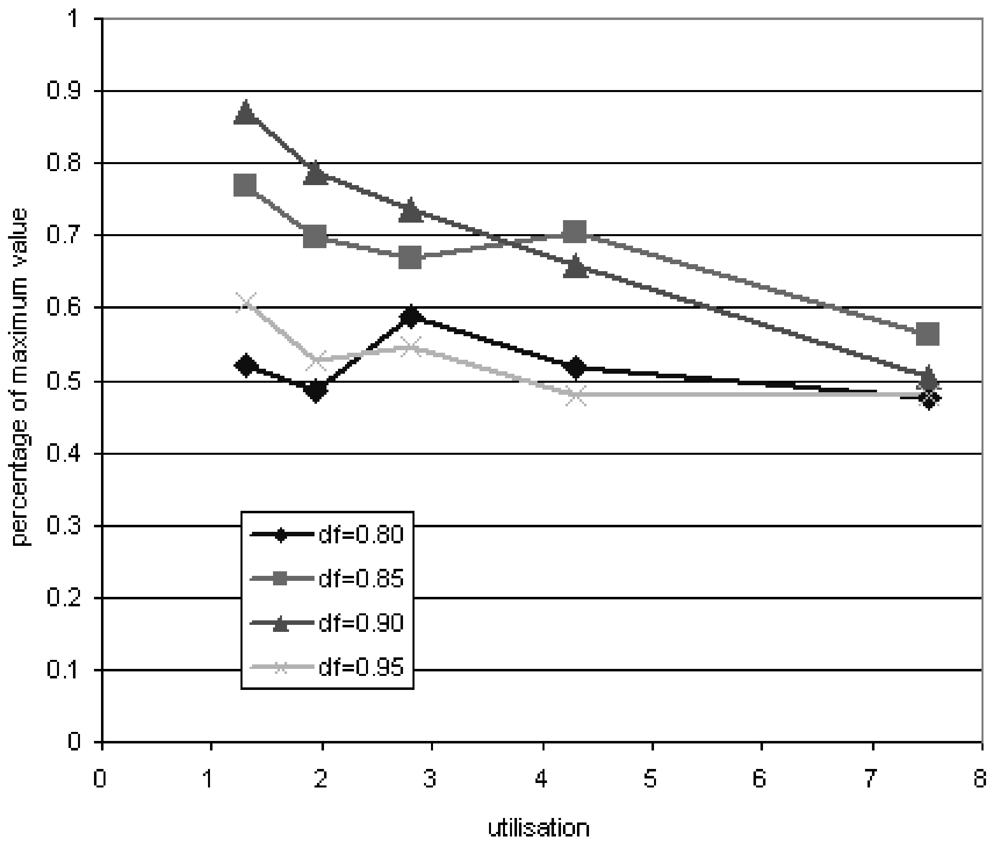


Figure 8.16: Utilisation - value diagram (decision-theoretic, nondeterministic release times)

We test a series of task sets with different characteristics in their utilisation levels, short and long task lifetimes according to their threshold deadlines, standard deviations of release times, etc.:

task set	#tasks	$std^*_{T_0}$	$U^{0.3}_{T_0}$	$U^{0.6}_{T_0}$	$U^{0.9}_{T_0}$
$\mathbb{T}_1$	10	0	0.81970	1.11281	1.34732
$\mathbb{T}_2$	10	1.65	1.21274	1.62199	2.01347
$\mathbb{T}_3$	10	2.23	1.74952	2.32516	2.88144
$\mathbb{T}_4$	10	3.98	2.62990	3.55418	4.42521
$\mathbb{T}_5$	10	5.48	4.65933	6.47665	7.66358

Although the influence of these and other parameters not mentioned in this chapter (e.g., homogeneity of period lengths) is complex, we can see that the scheduling allowance stabilises after a small number of scheduling phases for all these applications. We use the simulated annealing scheduler with a search ratio of 0.1 and a window size of 20, start with a scheduling allowance of 0.2 and set the controller parameters as follows:

C_p	C_i	C_d	spi	spd
0.3	0.6	0.1	5	5

Unfortunately, we are not able to give an analytical backing for the setting of the controller parameters. The setting we use was gained by experiment prior to the actual benchmark tests for the scheduling algorithms. The choice of parameters has to take into account properties of an application influencing the ability of the meta scheduling controller to stabilise, especially the frequency and regularity of task releases and the level of nondeterminism. Optimal controller parameters are therefore specific to an application. There is evidence that a PI controller rather than a PID controller could also be appropriate for the meta scheduling in our problem setting.

8.5.1 Stability

Starting from a scheduling allowance of 0.2, the simulations for different task sets show different tendencies to assume a certain value during the course of several scheduling phases (figure 8.17).

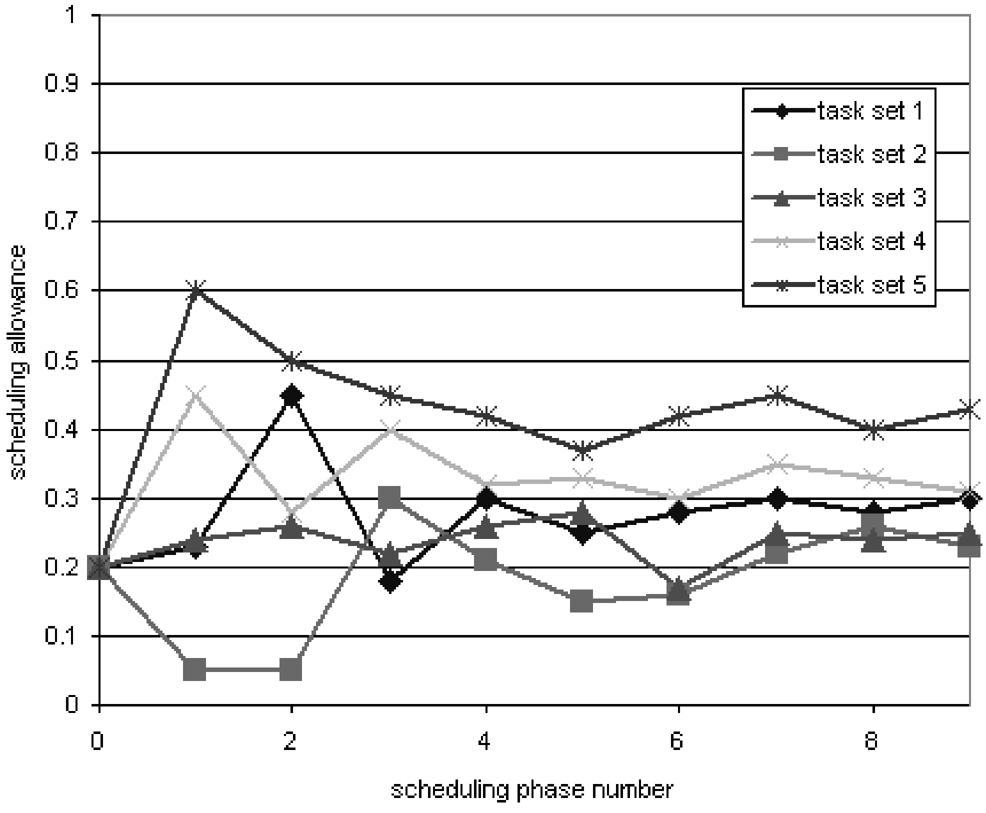


Figure 8.17: Flexible scheduling allowance with PID controller

The changes during the initial phases are in general bigger than in later ones. The ability of the system to stabilise to some extent depends on factors like the predictability of the release times, the load on the processor, and the size of the scheduling window.

8.5.2 Flexible vs Fixed Scheduling Allowance

The value of the schedules gained for a fixed percentage of the processing time reserved for the scheduler can be compared to the one achieved with the PID controller meta scheduling scheme.

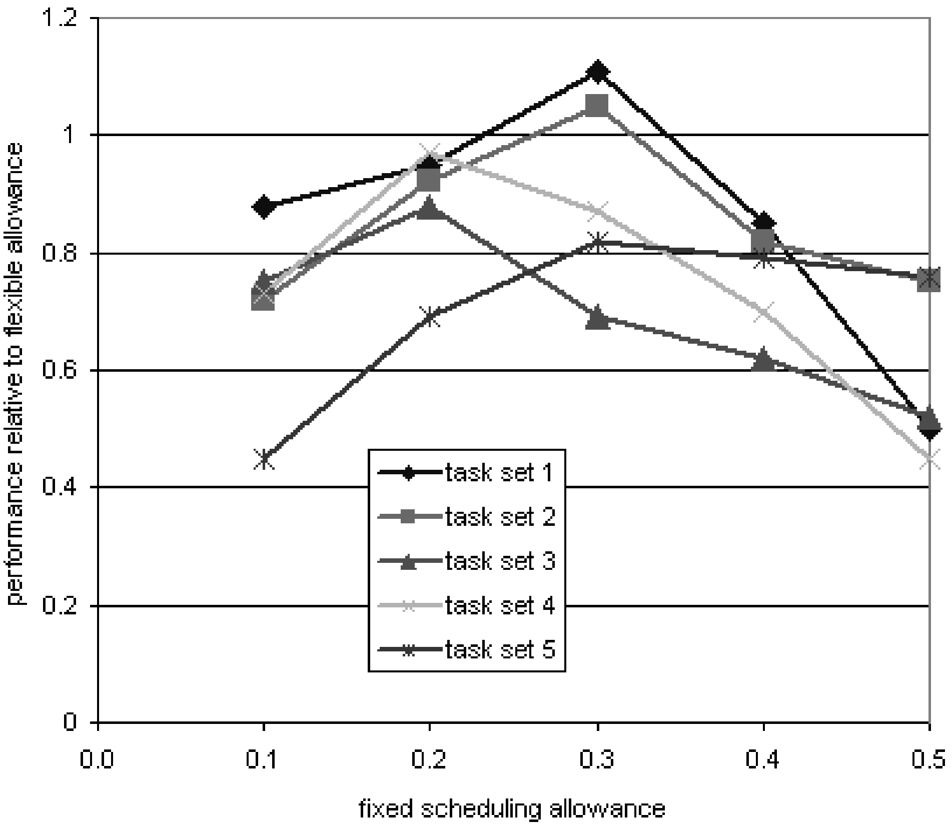


Figure 8.18: Comparison between fixed and flexible scheduling allowance

Figure 8.18 shows that in most cases, a fixed reservation cannot compete with the flexible scheme, because the time allocation to the scheduler is either too small to compute sufficiently good schedules or it reduces the cpu time available to application tasks too much. In some cases, the performance of a fixed reservation scheme is better than the one of the flexible scheme. However, the latter one is still preferable for several reasons. First, the ideal reservation is difficult to decide on before runtime. Second, the initial number of scheduling phases to find an optimal setting of the allowance is not significant in the long run. Third, the optimal allowance may change over time, so that no fixed reservation scheme can be appropriate.

Chapter 9

Scientific Context

I love deadlines. I love the whooshing sound they make as they fly by.
Douglas Adams

Work expands to fill the time available for its completion.
C. Northcote Parkinson

This chapter intends to give an overview of related work in the areas of both artificial intelligence and real-time scheduling research relevant for this thesis.

9.1 Flexible and Value-Based Scheduling

To distinguish flexible scheduling schemes from others with objectives defined implicitly via parameters like deadlines, period lengths, user-defined priorities, etc., the term *value-based scheduling* has been coined, being used for a wide variety of schemes with some *explicit notion of value* for tasks. As this explicit value is directly related to flexibility in scheduling schemes, we will use the terms *value-based scheduling* and *flexible scheduling* interchangeably.

9.1.1 Assignment of Value to Tasks

Before investigating the use of value in scheduling algorithms, a problem frequently neglected has to be mentioned, i.e., the finding of a suitable value assignment to the tasks of a given real-world problem.

The basic idea of value-based scheduling is to rate the tasks to schedule by a scalar real value or a real-valued function which simplifies comparisons between them. Ideally, a first step should be a formal proof that a value-based formulation for a given application exists. After that, typical relations between the tasks which can be derived from the application semantics are:

- preference: an asymmetric relation saying one object is more valuable than another one
- indifference: a reflexive and symmetric relation indicating two objects are comparable, but cannot be distinguished from each other (at least not with the data currently available)
- incomparability: an irreflexive and symmetric relation expressing that neither of the objects is more valuable than the other one

From these starting points, a complete ordering can gradually be inferred. Note that in general, above relations cannot be derived from the application completely and free of conflicts. Therefore, the next step is to eradicate inconsistencies in the preference relations between tasks. Consider the following importance relation on three tasks:

	T_1	T_2	T_3
T_1	$=$	$\leq$	$\geq$
T_2	$\geq$	$=$	$\leq$
T_3	$\leq$	$\geq$	$=$

Obviously, this relation cannot be expressed by any scalar real-valued attribute assigned to the tasks, because the relation is not transitive.

Burns et al. [BPB$^+$00] state that value functions should be cardinal, i.e., defined on all tasks and additive, so that a cumulative value can be calculated for the task set. Our definition of value functions for AND type nodes is derived from these ideas.

A further contribution of this work is the notion of alternatives for tasks, where the cumulative value of the task is defined as the maximum value of any alternative (figure 9.1). We used these ideas for the value functions of OR nodes in our model.

Burns et al. distinguish between several sources of the value attribute of a service, namely

- the quality of the output produced
- the time at which the task completes
- the history of previous invocations
- the condition of the environment

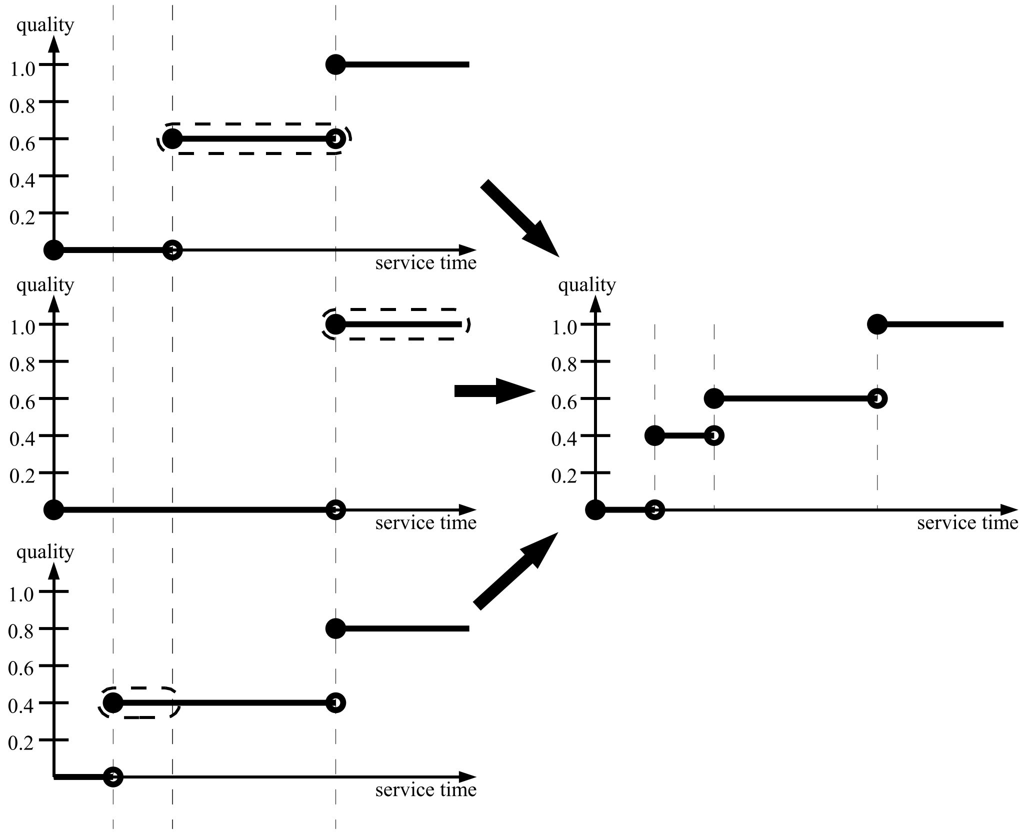


Figure 9.1: Cumulative value for set of alternatives

- the state of the computer system
- the importance of the task
- the completion probability of the task

The two sources of flexibility addressed in this thesis refer to the first two items in this list, so that we can rate relevant related work in several subcategories for each of the following directions:

Quality-flexible schemes owe their level of flexibility to the possible variations of quality, such that tasks may trade off quality for computation time and thus be able to meet their deadlines.

Timeliness-flexible schemes owe their level of flexibility to the fact that they do not have hard deadlines. Whereas the execution times of tasks in such schemes are fixed or unknown, the value of a computation may decrease gradually with later termination.

9.1.2 Quality Flexibility

In terms of quality, we distinguish between four levels of flexibility; these are as follows:

run-to-completion: Scheduling schemes relying on the traditional run-to-completion paradigm are not quality-flexible. The scheduler cannot influence the execution time or the quality of tasks, and scheduling decisions do not include these issues.

method selection: Method selection schemes are the simplest way of going beyond the run-to-completion assumption and trading off computation quality for execution time; they work on a set of run-to-completion tasks, of which one is to be selected by the scheduler. In order to facilitate finding valid schedules, one of these alternatives for every task should be a quick fall-back method performing the absolute minimum of computation for the task. Method selection schemes use the scheduler model of figure 1.2 with the execution parameters being the information which alternative of each task to choose.

contract schemes: Contract schemes require accurate prior knowledge of the relationship between certain parameters of the algorithms which implement individual tasks of an application, their service time requirements and the quality to be expected. The idea is to fine-tune the service requirements of a task by appropriately setting algorithm-specific parameters, e.g., repetition numbers for loops or the resolution for image processing algorithms. The simplest contract scheme allows a direct setting of the service times of tasks. Contract schemes rely on single implementations for each task, computations are not normally considered interruptible, and no intermediate results are gained when aborting them prematurely. They use the scheduler model of figure 1.2 with the execution parameters being the service times for each task and possibly algorithm specific parameters. Note that parameters cannot be changed once computation has commenced.

iterative refinement: In iterative refinement algorithms, the assumption is that a primary solution is available very quickly (conceptually at the very beginning), and a higher number of iterations improves on this initial solution. For example, so-called milestone methods produce intermediate results at pre-defined instants of time (milestones), and sieve functions generate new (presumably better) solutions by performing a series of operations (sieves) on them. Iterative refinement schemes use the scheduler model of figure 1.3.

9.1.3 Timeliness Flexibility

Just like for the category of quality flexibility, we distinguish between several levels of flexibility for the timeliness of tasks. These are:

no timing constraints: Scheduling schemes without explicit timing constraints include those outside the scope of real-time research as well as scheduling schemes guaranteeing real-time operations implicitly under certain conditions. Analytic results justify the usage of parameters other than timing constraints (user-defined priorities, period lengths, etc.) to model real-time applications.

scalar deadlines: Traditional deadline schemes are the simplest way to address timing constraints in applications directly. Several categories of deadlines have been described; scalar deadlines may be attributed properties like *hard* (missing the deadline for one task means failure of the entire application) or *firm* (missing the deadline for one task means it does not contribute to the performance of the application any more). We rate soft deadlines with a gradual decay of an explicit value for the task once the deadline has passed as belonging to one of the following two classes.

extensions to deadlines: Several models extending the simple notion of deadlines can be found, starting with certain kinds of soft deadline schemes. More general formulations of timing constraints than with deadlines can be achieved, e.g., by requiring only a certain percentage of deadlines to be held or by comparing time stamps of tasks with their estimated execution time to minimise the average lateness of tasks in an application. Models of this category allow for a limited level of flexibility in the timeliness of tasks.

utility functions: Real flexibility in the timing constraints of tasks can be achieved through explicit functions of the time, which we introduced earlier in this work as *utility functions*. As with deadlines, utility functions can be specified in absolute time or relative to the release time of tasks.

9.1.4 Classification

Figure 9.1 gives an overview of the combinations of flexibility classifications we could determine within existing scheduling schemes. The problem class of this thesis is primarily the one with highest flexibility in both directions (entry no. 16). Blank entries in the table indicate that scheduling schemes for the corresponding combinations could not be found. In the following sections, we will give examples for the entries in this table.

		no notion of quality	limited flexibility	limited flexibility	flexible quality
		run-to-completion	method selection	contract schemes	iterative refinement
no notion of timeliness	no timing constraints	▨ 1	▨ 2	3	▨ 4
no flexibility	deadlines	▨ 5	▨ 6	▨ 7	▨ 8
limited flexibility	extensions to deadlines	▨ 9	▨ 10	11	12
flexible timing constraints	utility functions	▨ 13	▨ 14	15	$\boxed{16}$

Table 9.1: Classification table for scheduling schemes

9.2 Run-to-Completion Tasks without Utility Functions

This section deals with scheduling schemes in the first column of table 9.1, specifically with entries 1, 5, and 9.

9.2.1 Models without Timing Constraints

This category comprises many kinds of simple scheduling algorithms like first-come-first-serve (FCFS), scheduling with user-defined priorities, or rate-monotonic scheduling (RMS). All of these algorithms make their decisions based on information other than quality levels or timing constraints. These parameters may be directly determined by the user or derived from properties of the task set (release order, period lengths). Depending on analytic results available for individual algorithms, they are generally rated as belonging to the category of real-time or non-real-time scheduling algorithms, respectively. In general, e.g., RMS is considered a real-time algorithm[LL73] (and is, in fact, one of the most wide-spread real-time scheduling algorithms due to its simplicity), whereas FCFS is not considered a real-time algorithm. These classifications are based upon data like the maximum latency of tasks, a measure for the suitability of scheduling algorithms for real-time computations. This class of algorithms can be found in entry no. 1 of the classification table 9.1.

9.2.2 Deadline Models

The most straight-forward way to introduce explicit timing constraints into the development of real-time scheduling algorithms is by means of deadlines, i.e., the latest time when the execution of individual tasks must be finished. Starting from the pioneering work by Liu and Layland [LL73], deadline scheduling schemes have been investigated thoroughly. Well-known examples of this class of scheduling algorithms are earliest-deadline-first (EDF) and least-slack-time-first (LST), with priorities determined exclusively or at least partially by deadlines. Deadline schemes are frequently further divided according to the consequences of missed deadlines. For example, missing hard deadlines usually means a complete failure of the entire application, whereas a missed firm deadline merely means that the task does not contribute to the overall performance of the application any more. In our classification table 9.1, this class of scheduling algorithms can be found at position 5.

9.2.3 Extended Deadline Models

This section describes a collection of extensions to deadline scheduling schemes which we rate into category 9 of table 9.1.

9.2.3.1 Discrete Utility Models

McElhone and Burns [MB00] oppose complex concepts of time-dependent utility; they argue that these are unsuitable for real-time systems, especially if scheduling is to take place dynamically and concurrently on the same processor as the application tasks. Instead, they attempt to develop a simplified computational model which is rich enough to allow complex requirements to be mapped onto it on the one hand, but itself requires minimal run-time support on the other hand. Tasks belong to one of five types with different scalar utilities (values), abortability classifications (e.g., not abortable for mandatory tasks, abortable before start for medium tasks and abort at any time for low utility tasks), execution time specifications (bounded or unbounded), schedulability guarantees (online or offline), and deadline types (hard, firm, soft).

9.2.3.2 End-to-End Deadlines

End-to-end deadlines pose timing constraints at higher-level objects (*work items*) than on low-level tasks. The approach traditionally taken is to break down an application into atomic tasks and derive timing requirements for these tasks (*artifact deadlines*) from the actual timing requirements of the application (*specification* or *end-to-end deadlines*) at design time. Scheduling and schedulability analysis then takes place on the basis of such a set of tasks with associated deadlines. However, it has been noted that a certain degree of flexibility might be lost due to assignment of possibly inappropriate or too conservative deadlines.

Goforth et al. [GHWB95] advocate the opinion that schedulability analysis should be part of the design process and not be done on a set of atomic tasks. The application itself is responsible for prioritising these work items, not a general-purpose scheduler without any knowledge of the semantics of a specific application. This way, the application has control over which parts of an application may be discarded in case of overload.

The advantage of not giving up the information provided by the original requirements specification at an early stage could be shown to be especially beneficial in the area of real-time networking, as [RKJZ99] demonstrates with an ATM network traffic scheduling scheme; in fact, the term end-to-end constraints has its roots in the research on real-time networks. An interesting multimedia application platform making explicit use of end-to-end constraints was developed in [NPB02].

9.2.3.3 Window Constraints

Another means of flexibility in the timeliness of real-time applications is relaxing the requirement of every single task instance to finish before its deadline. Instead, it is sometimes sufficient to demand that at least m out of any window of k consecutive instances of a task can meet their deadlines. *Window constraints*, *weakly hard constraints* and *(m,k)-firm deadlines* are different names for the same concept.

As an example, consider one of the algorithms presented by Ramanathan et al. in [Ram97, Ram99, HR95]; it uses the following function to guarantee that m out of k instances of a task with (m,k)-constraint are classified as mandatory, i.e., they must meet their deadlines:

$$\text{task instance } i \text{ is} \begin{cases} \text{mandatory} & \text{if } i = \left\lfloor \frac{k}{m} \cdot \left\lceil \frac{(i-1)\cdot m}{k} \right\rceil \right\rfloor + 1 \\ \text{optional} & \text{otherwise} \end{cases}$$

The weakly-hard real-time model by Bernat et al. [Ber98, BBL01, BB97] introduces tolerance functions defining maximum times for task instances to run beyond their deadlines.

Balbastre et al. [BRC02] present a model for (m,k)-firm deadlines such that the constraint is fulfilled if the computation time reaches a minimum level for all tasks and can be allowed a certain extra amount for m out of k consecutive task instances. The most interesting result of this work is an upper bound for the extended computation time which can be awarded to the tasks without jeopardising the schedulability of a task set.

Mittal et al. [MMM00] attempt to schedule hybrid task sets consisting of hard real-time and quality-of-service degradable (primarily (m,k)-firm) tasks composed of mandatory and optional parts on multi-processor systems using a set of simple admission control and quality adaptation heuristics.

9.2.3.4 Clock-Based Scheduling

The approach taken by clock-based scheduling algorithms is to make decisions according to task-local times advancing only when the task executes on a processor. The primary aim of clock-based scheduling was traffic control in networks [Zha91]. However, it has also successfully been applied to processor scheduling. The major advantage of clock-based algorithms is that they allow for a very natural modelling of mixed sets of real-time and non-real-time (best-effort) tasks. Whereas for real-time tasks the run-to-completion assumption applies and timing constraints may be flexible to varying degrees, the computation time of best-effort tasks is flexible. Hence, this group of scheduling algorithms can be classified in entries 4 and 9/13 of table 9.1. However, the problems are distinct from quality / utility scheduling, as the two aspects of flexibility never apply to the same task in clock-based scheduling algorithms.

The BERT (for Best-Effort and Real-Time) scheduler by Bavier et al. [BPM99, BP00] is derived from the simple idea of proportional share scheduling, which allocates some percentage of computational resources to individual tasks. BERT uses virtual clocks and a simple grid of four different classes of tasks (*unimportant real-time*, *unimportant best-effort*, *important real-time* and *important best-effort*) to express criticality and importance of tasks. Under certain conditions, real-time tasks may steal CPU time from others to meet their timing constraints.

The BVT (Borrowed-Virtual-Time) model by Duda and Cheriton ([DC99]) is more complex than the BERT scheduler, as it works with a potentially infinite number of dynamically assigned priorities instead of a small number of task classes. This means a more fine-granular modelling of a problem, but also incurs higher computational overhead.

9.2.4 Evaluation and Correlation to this Thesis

Survey publications like [Liu00] and [Che02] cover many of the research directions in both dynamic and static priority scheduling schemes like EDF, RMS, or the models we collectively referred to as extended deadline models.

Scheduling schemes for tasks without timing constraints are not within the focus of this work. Deadlines can be modelled easily in our scheme, but the simple scheduling heuristics mentioned above did not influence the more complex algorithms in this thesis. Even though interesting theoretic results have evolved from this class of algorithms and their simplicity of implementation makes them easy to apply to many real-world problems, these schemes are not ideally suited in contexts where assumptions like independence of tasks and a-priori knowledge of the execution times do not hold. Furthermore, myopic simple scheduling heuristics like EDF tend to show drastically bad performance in case of transient or permanent overloads; in many contexts, violation of timing constraints can be tolerated to some degree, as long as the system performance degrades gracefully at high load.

End-to-end deadlines gave the incentive for the hierarchical task model of this thesis, where timing constraints can be imposed at any level. Just as in the work on end-to-end constraints cited in this section, we try to lose as little of the flexibility of the original problem as possible during the design process by specifying timing constraints on as high a level as possible. Window constraints cannot be expressed in the quality / utility scheduling model; we nevertheless included them as an important kind of flexible timing constraints outside the limits of our work. We use the idea of local time from clock-based scheduling schemes for the notation of quality functions and the value functions derived from them; we do not, however, adapt the simple heuristics for scheduling algorithms from these clock-based schemes.

9.3 Models with Limited Quality Flexibility

This section deals with scheduling schemes in the second and third column of table 9.1, specifically with entries 2, 6, 7, and 10.

9.3.1 Method Selection Schemes

In method selection schemes, the scheduler decides on an alternative for each task; execution times and quality of the alternatives must be known or at least assessable with sufficiently high accuracy.

9.3.1.1 Task Pair Scheduling

Streich et al. [Str94, GKS95] describe a model they call task pair scheduling. The motivation behind task pair scheduling is that tasks with complete a-priori knowledge on worst-case execution times and rather exact knowledge on the release times are not realistic for many applications.

Streich at al. use an upper bound on execution time for a certain high percentage of invocations, e.g., 90%. This parameter is called optimistic-case execution time (OCET). In task pair scheduling, each real-time task is represented by a soft and a hard task. If the scheduler is able to reserve enough resources to execute the soft task, the soft task is chosen for execution. Otherwise the hard task is selected. The system has to ensure that at least all hard tasks can be executed in a timely manner. The scheduler needs to know the worst-case execution times of all hard tasks and the optimistic-case execution time of all soft tasks. Deadlines are specified for task pairs and apply to the soft task and hard task components alike.

Several properties distinguish task pair scheduling from imprecise computation scheduling with mandatory and optional components (see below):

- The successful termination of the hard task is not a precondition for the execution of the soft task.
- Hard and soft tasks are not computationally correlated, i.e., executing the soft task for some time does not reduce the execution time of the corresponding hard task.
- Task pair scheduling is optimistic inasmuch as it executes the soft task if the probability that it can finish before its deadline is sufficiently high.

The scheduler which is suggested for task pairs runs soft tasks in round-robin fashion; obviously these tasks have to be preemptive. Hard tasks, on the other hand, are non-preemptive, must be guaranteed offline and are run as late as possible according to their deadline and worst-case execution time, but with higher priority than any soft task. Task pair scheduling works on tasks with and without deadlines and belongs to both entries 2 and 6 of table 9.1.

9.3.1.2 Design-to-Time Model

The design-to-time model was presumably introduced by D'Ambrosio [D'A89], and the term seems to have been coined by Bonissone and Halverson [BH90]. Our description of design-to-time scheduling was largely taken from the extensive work by Garvey and Lesser [GL93, Gar96, GL96b, GL96a]. Design-to-time scheduling assumes that in general multiple methods with different execution times and quality values are available to implement individual tasks of an application, so that the tradeoff between solution quality and the timeliness of computations can be exploited. Hence, it clearly falls into the method selection category of models.

Although there is a finite number of alternatives (called methods) available for each task, these alternatives do not necessarily represent only distinct points in the service time / quality plane; in other words, alternatives may be described by stochastic distributions with regard to both quality and service time requirements. Design-to-time scheduling can handle both soft and hard deadlines. The additional overhead incurred through method selection pays off primarily in overloaded systems, i.e., in cases when it is not possible to execute the optimal solution for all tasks.

Even though different algorithms for the same problem are not always available, alternative implementations can frequently be derived from the original one by using one or a combination of the following operations:

- approximation of input or intermediate data
- use of approximative instead of optimal (sub)algorithms
- skipping optional steps

Garvey and Lesser devised a controller architecture for dynamically allocating resources to tasks at a high level and a micro-scheduling unit to execute the tasks at low level. A feedback mechanism from the execution subsystem to the high-level controller may trigger rescheduling if necessary because of inaccurate predictions or unexpected events. The scheduling algorithm is executed every time the low-level subsystem triggers it; the algorithm is repair-based. i.e., it starts with the best-quality alternative of every task and decreases service levels until the schedule becomes feasible.

In recent years, design-to-time scheduling has evolved into a new model called design-to-criteria scheduling [WL00], taking into account not only time, but also more general data. Design-to-time and design-to-criteria schedulers can be classified to numbers 6 and 10 of table 9.1.

9.3.2 Contract Schemes

In contract schemes, the scheduler must decide on appropriate parameters for each task; an accurate knowledge of the influence of the parameters on execution time is essential. The simplest contract scheme is to set the execution times for all tasks directly.

9.3.2.1 RTA$*$ Search

Korf adapted the A$*$ search algorithm for real-time operation; he called the real-time heuristic search algorithm RTA$*$ [Kor85, Kor87, Kor88, Kor90]. Whereas the average-case and worst-case time needed for the original A$*$ search algorithm may differ significantly, the execution time of the modified algorithm can be controlled much better. The basic idea is to limit the search to a finite horizon and apply a pruning mechanism on the search tree, so that the algorithm can commit to action in constant time. This algorithm can be executed for any given search horizon (and hence, in any given computation time), but useful intermediate results are not generated. Although RTA$*$ is a rather specific kind of algorithm, it can be seen as a forerunner of more general contract schemes.

9.3.2.2 Contract Anytime Algorithms

Russel and Zilberstein [RZ91, Zil93] suggested the concept of contract anytime algorithms. Contract anytime algorithms can be allocated an arbitrary computation time and are guaranteed to produce a reasonable result for any such allocation. However, they must know in advance (i.e., before they are scheduled for execution) how much of the computational resources are available to them. If they are terminated prior to the execution time for which they were initially intended, they need not achieve any positive quality. To distinguish them from the anytime algorithms described by Boddy and Dean (see below), Russel and Zilberstein call the latter ones *interruptible anytime algorithms*.

Every interruptible anytime algorithm can be trivially interpreted as a contract algorithm by simply discarding the result of the computation if it is interrupted prematurely. On the other hand, Zilberstein [Zil93] showed that a contract anytime algorithm can be transformed into an interruptible anytime algorithm such that it achieves at least the same quality for any allocation of computation time as the original algorithm if awarded four times the original allocation. Figure 9.2 shows this relationship.

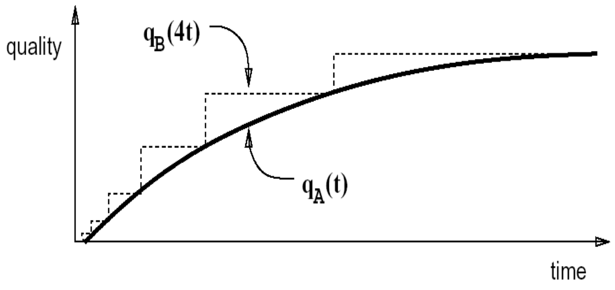


Figure 9.2: Performance profiles for interruptible and contract anytime algorithms

In their work, Russel and Zilberstein suggest to create a new contract anytime algorithm from a set of (contract and interruptible) anytime algorithms together with a deadline. They develop a methodology called *local compilation* the purpose of which is to decide how much time within an interval each of the original anytime algorithms is awarded. Local compilation can only be performed on sets of independent tasks or of tasks with a linear or tree-structured precedence graph. For more general directed acyclic precedence graphs, Russel and Zilberstein have to resort to heuristics.

We can find contract schemes only in entry 7 of table 9.1, because they intrinsically always need deadlines associated with tasks.

9.3.3 Evaluation and Correlation to this Thesis

Method selection has been incorporated into quality / utility scheduling via the *and*/*or* hierarchy of tasks. As method selection in our model is part of the overall resource allocation algorithm, we cannot make use of the heuristics of the schemes mentioned in this section. We do, however, adapt from the design-to-time model the various sources of flexibility in the time / quality trade-off (approximation, skipping of optional steps, etc.) and map these into our quality functions during the design process. With the theorem on contract and interruptible anytime algorithms, we can incorporate contract anytime tasks into the quality / utility scheduling model, which uses interruptible anytime tasks.

9.4 Timeliness-Flexible Schemes

Instead of using hard deadlines, more flexibility can be achieved by employing functions of the time (*not* the execution time of tasks) to express a more fine-granular notion of urgency. All of the approaches in this section are based on the run-to-completion assumption, and their source of flexibility lies in the continuous utility function rating timeliness of tasks.

9.4.1 Repair-Based Best-Effort Scheduling

Locke and Jensen [JLT85, Loc86] developed a first model employing time-dependent utility functions (which they call *value functions*). In this model, tasks are assumed to have arbitrary release times, so that the additional information on the easier dynamics of periodic tasks is not exploited. Tasks can be started, preempted and resumed at any time after their release time. A dynamic scheduler is responsible for making decisions on which tasks to execute whenever a new task is released or when a task terminates. Precedence constraints are not treated explicitly in this model; it is assumed that tasks are only released to the system and hence to the scheduler when all of their precedence constraints are satisfied. Apart from the release time, further attributes of a task are its expected execution time, its deadline and a value function. All tasks are supposed to be resident in memory, which is a common assumption in real-time computing, and the target architecture is a shared-memory multi-processor system. The value function is usually defined such that it has some discontinuity at the deadline or is at least not differentiable in this point, as can be seen in the example value function of figure 9.3, which has a linear decrease prior to the deadline and an exponential decrease after the deadline.

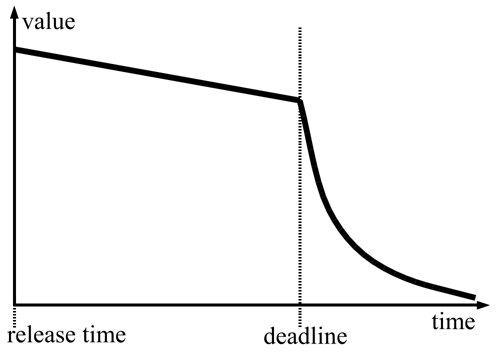


Figure 9.3: Best-effort value function

Specific value functions used in the simulation work of Locke are of the form

$$V(t) = \begin{cases} K_1 + K_2 \cdot t - K_3 \cdot t^2 + K_4 \cdot e^{K_5 \cdot t} & \text{if } t \leq deadline \\ K_1' + K_2' \cdot t - K_3' \cdot t^2 + K_4' \cdot e^{K_5' \cdot t} & \text{if } t > deadline \end{cases}$$

The objective function to drive the scheduler is simply the sum of the values of all finished tasks (or, rather, the tasks assumed to finish before some point in the future). As optimal solutions to this scheduling problem are usually intractable, heuristic methods are employed, based on two perceptions:

- On a single processor, a set of tasks with precisely known release times, execution times and deadlines, earliest-deadline-first scheduling (i.e., scheduling according to increasing deadlines) is known to be optimal, as long as the utilisation never exceeds 1.
- On a single processor, a set of tasks with precisely known release times and execution times and scalar utility values awarded to them when they finish, value-density scheduling (i.e., scheduling according to decreasing values of $\frac{utility}{execution\ time}$) is optimal.

The repair-based scheduling algorithm proposed (Clark calls it LBESA for Locke's Best-Effort Scheduling Algorithm) can be outlined as follows: First, tasks are ordered by increasing deadlines. If all value functions have constant values prior to their deadlines and the processor is not overloaded, the schedule is known to be optimal. For all other value functions, it is assumed that the decrease in value prior to the deadline of a task is relatively small, so that for non-overloaded systems it can still be expected that the schedules gained are close to optimal.

Overloads cannot be determined with certainty, as the execution times cannot be taken for granted. If the probability of an overload exceeds a given threshold, the overload is counteracted by removing tasks according to a predefined heuristic. Some of the heuristics suggested are

- static value density (Locke): remove task with minimal $\frac{value(t)}{computation\ time}$
- dynamic value density (Aldarmi and Burns [AB99]): remove task with minimal $\frac{value(t)}{remaining\ computation\ time}$
- strongly dynamic value density (Aldarmi and Burns): remove task with minimal $\frac{value(t)}{(remaining\ computation\ time)^2}$
- strongly dynamic timeliness density (Aldarmi and Burns): remove task with minimal $\frac{value(t-computation\ time)}{(remaining\ computation\ time)^2}$
- BE-h (Mossé et al. [MPR99]): two-stage static value density; classify tasks into long-runners and short-runners, remove long-runner with minimal value density if there is any, remove short-runner with minimal value density if there is no long-runner
- BE-v (Mossé et al.): find the tasks with lowest value density (lvd) and lowest value (lv), remove lvd if $\frac{value\_density(lv)}{value\_density(lvd)} \geq \frac{expected\_utility(lv)}{expected\_utility(lvd)}$ and lv otherwise
- simplified-rolling-horizon-N$_{wt}$-rule (Morton and Pentico [MP93]): define the partial makespan as the sum of all execution and idle times up to the first tardy task (with the expected completion time being after its deadline); remove the task with the smallest ratio $\frac{current\ value}{(partial\ makespan\ for\ task\ set\ including\ the\ task)-(partial\ makespan\ for\ task\ set\ without\ the\ task)}$

The heuristics of Mossé et al. aim at reducing the problem that tasks with short execution times are not treated favourably by the static value density mechanism in the original work by Locke, because the processor reservation for them is easily swallowed up by minor fluctuations in the processor load. Aldarmi and Burns want to avoid the effect that tasks are aborted shortly before their completion, which can easily happen with Locke's algorithm. The objective of the heuristic by Morton and Pentico is to remove a task with little contribution to overall performance and high potential of reducing lateness of other tasks.

Tokuda, Wendorf et al. [TWW87, Wen88] investigated the problem of deliberation costs for best-effort scheduling performed concurrently on the same processor as the application tasks and demonstrated the performance limitations of this scheduling algorithm.

Clearly, best-effort scheduling belongs to entries 13 and 14 of table 9.1, as it incorporates both finding appropriate time intervals to execute the tasks and method selection by discarding component tasks of an application.

9.4.2 Constructive Scheduling with Time-Value Functions

Chen and Muhlethaler investigate the problem of scheduling a set of non-preemptive tasks with an associated *time value function* (i.e., a function of time like the value function in the best-effort model) on a single processor [CM96, MC92]. They assume tasks to be in one of several phases depending on the time (*unavailable*, *available*, *optimal*, *available* and *dead*) and suggest to model these phases within one function of the time, as demonstrated in figure 9.4.

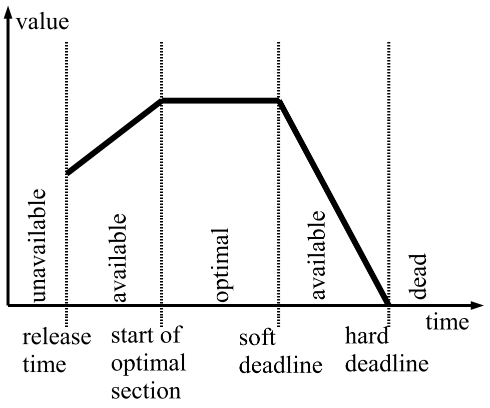


Figure 9.4: Time value function

Figures 9.5a), 9.5b), and 9.5c) show examples of time-value functions describing hard, firm, or soft deadlines, respectively. Note that the arrow in figure 9.5a) indicates that the value associated with a task having passed its hard deadline is $-\infty$. Figure 9.5d) shows the time-value function of a task with steeply ascending or descending edges at the borders of the positive-valued interval. Examples can be found in multimedia applications, where an early display of a video frame is considered as bad as a late display.

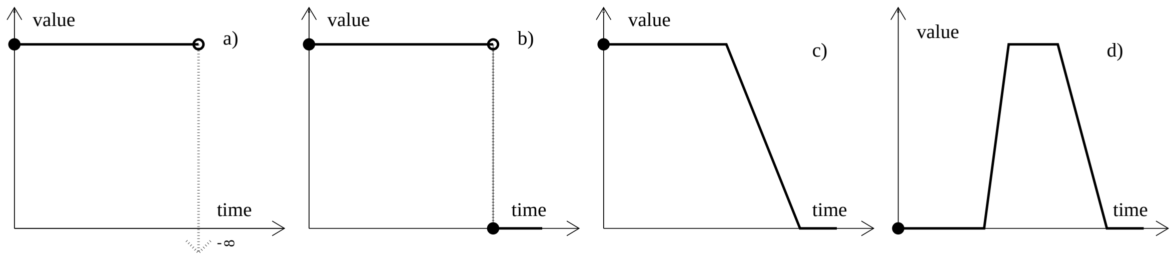


Figure 9.5: Typical time-value functions describing deadlines

Unlike Locke with his repair-based model, Chen and Muhlethaler propose a constructive scheduler. Let $\mathbb{T}$ be a set of n tasks released at time 0 with execution times $p_1, \ldots, p_n$

and time value functions $f_1, \dots, f_n$. Provided that the processor is never idle before all tasks have finished, a schedule for the set of non-preemptive tasks consists of a sequence of tasks $\sigma = (\sigma(1), \dots, \sigma(n))$. For a given sequence σ, the finishing time of task $\sigma(k)$ is $t_k := \sum_{i=1}^{k} p_{\sigma(i)}$, and the objective of the scheduler can then be expressed as finding a sequence such that the sum of the values gained is maximal among all possible sequences:

$$\max_{\sigma} \sum_{k=1}^{n} f_{\sigma(k)}(t_k)$$

The approach taken to solve this ordering problem is to partition the task set such that optimisation can be performed locally on the task subsets. For this purpose, a time-dependent precedence relation is introduced between tasks. A task i is said to *precede* task j at time t if the objective function evaluates to a higher value for

$$\sigma = (\sigma(1), \dots, \sigma(k), i, j, \sigma(k+3), \dots, \sigma(n))$$

than for

$$\sigma = (\sigma(1), \dots, \sigma(k), j, i, \sigma(k+3), \dots, \sigma(n)).$$

Task i is said to *strongly precede* task j at time t if i precedes j at time t' for all $t' \geq t$.

Chen and Muhlethaler were able to prove that all optimal sequences for the task set are concatenations of optimal sequences for the task subsets gained by partitioning the set according to the strong precedence relation. If there is an irreducible optimal decomposition, then it is unique. The problem of finding optimal sequences has now been transformed into the problem of finding the optimal irreducible decomposition and subsequently optimal sequences for the subsets of the task set. Both heuristics and optimal local optimisation algorithms for several objective functions are presented in the work of Chen and Muhlethaler. Discarding tasks is not possible in this model, so that the sorting algorithm suggested belongs to entry 13 of table 9.1.

9.4.3 Evaluation and Correlation to this Thesis

Utility functions like the ones described in this section are used in all quality / utility scheduling problems. Both best-effort scheduling and the constructive mechanism provide ample examples. However, the non-preemptive model by Chen / Muhlethaler is not very closely related to quality / utility scheduling. On the other hand, the computational model introduced by Locke is very similar to ours. We also use some of the heuristics for default actions in the MDP-based scheduler, when calculating more accurate strategies (i.e., defined on more tasks) cannot be afforded.

9.5 Quality-Flexible Schemes

In iterative refinement scheduling schemes, the scheduler has to decide on the amount of service time allocated to each task on the basis of continuous monotonically increasing performance profiles or quality functions.

9.5.1 Anytime Algorithms

The origin of *interruptible anytime algorithms* (*anytime algorithms* for short) can be seen in the context of application-specific subsystems to introduce real-time requirements into planning systems. Dean and Boddy [DB88, BD89] presented the notion of anytime algorithms, where for an arbitrary execution time, a reasonable result can be received from the algorithm. Rising computation time results in a higher or equal quality of the computation, such that a performance profile can be derived which is monotonically increasing (and convergent) with the service time of a task. Underlying iterative refinement algorithms (e.g., heuristic search or dynamic programming algorithms) have, however, been studied prior to the introduction of the general concept of anytime algorithms. Due to the original area of application, Dean and Boddy refer to their own ideas as *time-dependent planning problems*. The term *anytime algorithms* arose from later works on deliberation scheduling, the explicit reasoning on the cost of calculating plans ([Bod91, BD94]).

Zilberstein [Zil93] distinguishes between various quality metrics for anytime algorithms, of which these are regarded as most useful:

certainty: The metric of certainty can be used for classification problems. Imagine objects must be classified as belonging to one of several categories; a certainty metric indicates the probability that the result of classification is correct. As more and more evidence can be collected over time backing an assumption of class membership, the level of certainty is a function of computation time.

accuracy: The accuracy of results means the distance of results from an exact answer. Accuracy metrics define a error term decreasing with computation time, applicable, e.g., to the position detection of autonomous robots with the error being the distance of the estimated position to the actual current position.

specificity: For certain algorithms, although a computation always yields correct results, an increase in computation time may still mean an increase in quality, namely by increasing detail, for example through a higher image resolution in image-processing software. In these cases, the specificity metric is frequently employed.

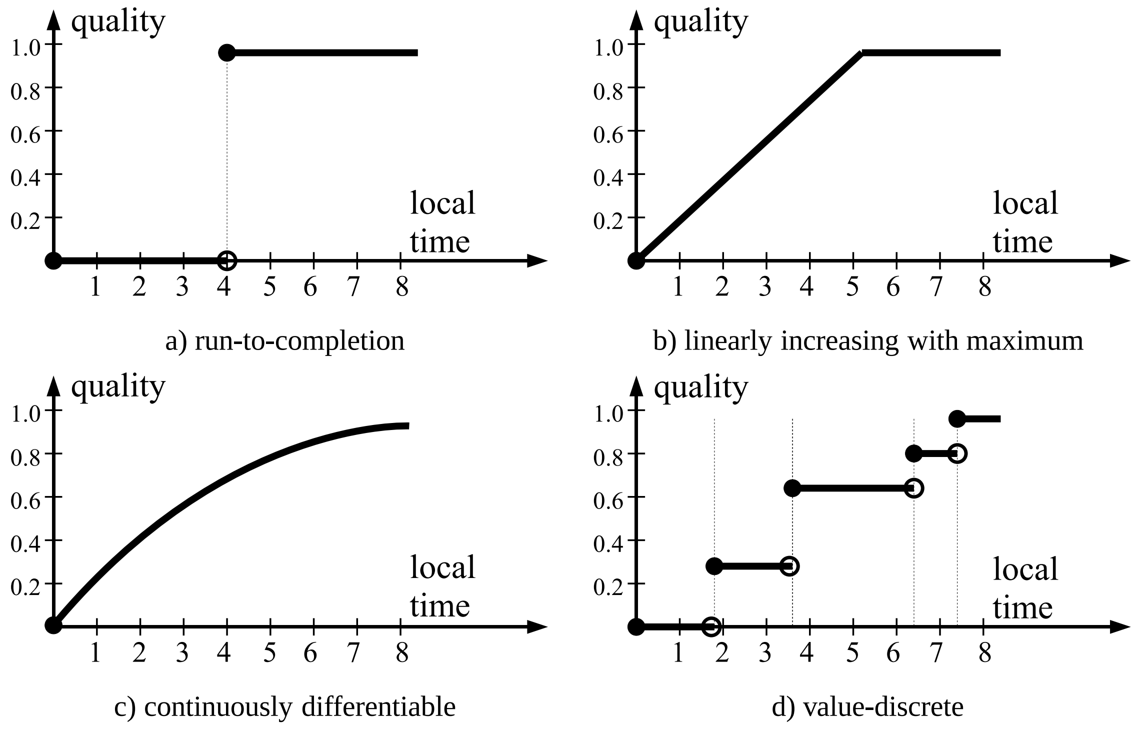


Figure 9.6: Typical performance profiles describing anytime behaviour

Figure 9.6a) shows the performance profile for a task with fixed execution time, figure 9.6b) a linearly increasing performance profile with maximum value, figure 9.6c) a continuously differentiable performance profile, and figure 9.6d) a value-discrete (piecewise constant) performance profile. Figure 9.7 compares a set of design-to-time quality-time tradeoffs to the performance profile of an anytime algorithm.

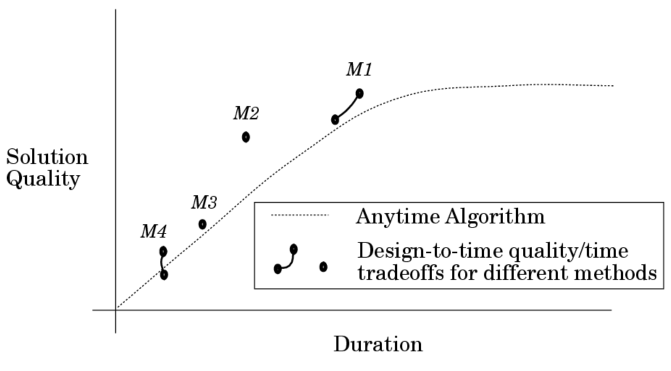


Figure 9.7: Design-to-time vs anytime algorithms

Zilberstein introduces three types of performance profiles instead of the single one in the original concept; this facilitates coping with the uncertainty of information available to the scheduler. These types of performance profiles are:

expected performance profile: The expected performance profile maps the computation time awarded to an anytime algorithm to the expected quality of the results. This is the kind of performance profile introduced by Boddy and Dean [BD89, DB88] and Horvitz [Hor87, Hor88] (see below), and it is especially useful if the variance of the expected quality is small or even zero, as in this case it offers very accurate (or even complete) information on the performance.

performance distribution profile: The performance distribution profile of an algorithm is a function that maps computation time to a probability of the quality of the results. Hence, it offers a more general description of the performance. Performance distribution profiles are, however, more difficult to be gained and to be evaluated. Their application is recommended for larger values of variance in the expected quality.

performance interval profile: The performance interval profile maps the computation time of an algorithm to the upper and lower bounds of the quality of the results. This can be seen as a compromise between the expected performance profile and the performance distribution profile: Performance interval profiles offer a compact representation and in general a sufficiently good estimate of performance distribution profiles (the performance distribution profile is replaced by a linear approximation).

Anytime algorithms have been used in schedulers with or without deadlines and are therefore rated into entries 4 and 8 of table 9.1.

9.5.2 Flexible Computations

Simultaneously, but independently from Dean and Boddy, Horvitz developed a model for activities of an application called *flexible computations* in a series of publications [Hor87, Hor88, EG91]. This model is very closely related to anytime algorithms. Focuses of this work are practical flexible implementations for existing real-world problems and complexity issues especially for traditional and flexible sorting algorithms.

Horvitz calls performance profiles *value functions* and defines their properties as:

value continuity: Value functions are surjective functions into the continuous interval [0;1].

value monotony: Value functions are monotonically increasing with service time.

convergence: Value functions converge to an optimal value.

Flexible computations are classified into categories 4 and 5 of table 9.1, just like anytime algorithms.

9.5.3 Imprecise Computations

One decisive property of flexible scheduling schemes from real-time computing research is that they employ a two-stage strategy: the primary goal is to guarantee all (hard) deadlines (usually offline); maximising some kind of overall value is only the secondary goal, and trading off missed deadlines for higher value is never an option [BB01]. Hence, a certain minimum service level is usually required from all tasks in order to classify a schedule as feasible. In this regard, flexible real-time scheduling schemes combine the solution strategies of satisficing and optimising, whereas artificial intelligence scheduling schemes usually employ purely optimising techniques and do not require a minimum service level, so that it is not possible to guarantee hard constraints.

The basic model of imprecise computations is as follows: Tasks have a known worst-case execution time, a deadline, and possibly a positive weight to express a relative importance of the tasks. Tasks in the imprecise computation model are defined to be preemptive. They consist of a mandatory and an optional part, and the optional part cannot start execution before the mandatory part has finished. In imprecise computation scheduling, a task is called *completed* if its mandatory part has been assigned sufficient units of processing time. If a task was able to complete its optional part, it is called *precisely scheduled*; otherwise it is called *imprecisely scheduled*. A schedule is called *precise* if all the tasks are precisely scheduled, and *complete* if all mandatory parts of tasks can be executed. A scheduling algorithm for imprecise computations is called *optimal* if it always finds a precise schedule whenever it exists, and a complete, but imprecise schedule with maximum value whenever a complete, but no precise schedule exists. The quality of optional computations is expressed by a monotonically decreasing error function or a monotonically increasing reward function.

The error function for an individual task is a function of the distance of the computation time awarded to the optional part of a task and its (worst-case) execution time:

$$\epsilon_i = E_i(o_i - \sigma_i)$$

where o_i is the execution time of the optional part of the i-th task and σ_i the service time allocated to this task. The simplest (and probably most widely used) definition for the error function is this distance itself:

$$\epsilon_i = o_i - \sigma_i$$

Liu et.al. [SLC89, SLC92, LLS$^+$91] describe several definitions of error functions for task sets, calling them *performance metrics*, and investigate the existence of solutions for the corresponding scheduling problem on a single processor.

- minimisation of weighted sum, maximum, or arithmetic mean of the task errors
- minimisation of number of discarded optional tasks
- minimisation of the number of tardy tasks (optional tasks exceeding some acceptable error)
- minimisation of the average response time

Burns, Bernat et al. [BB01, BB02, BBB02] propose a model for scheduling systems that guarantee hard deadlines for mixed sets of periodic, aperiodic, and sporadic tasks and use spare resources to maximise total system quality. One degree of freedom to address is the strategy when to use slack times (intervals of time when processors are not reserved for the mandatory part of any task) to schedule optional parts for execution. Probably the most common approach is *eager slack usage*, i.e., to make slack available for running non-hard components as soon as it is available. Eager slack usage means that mandatory parts of tasks are delayed as far as possible in order to be able to execute optional parts as soon as possible. Eager slack usage is not optimal, because the low-value optimal parts of the task may delay the mandatory part of a second task and ultimately prevent the high-value optional part of the second task from execution. However, the opposite strategy, namely *lazy slack usage*, can have similar deteriorating effects on the performance. In this case, mandatory parts are always executed as soon as possible, and all optional parts are run as background tasks. This may lead to unnecessary idle times of the processor and hence suboptimal schedules. Both suboptimal simple heuristics and standard optimisation techniques are frequently used for scheduling flexible computations.

Periodic task sets are an important special case with strong connections to practical real-time applications and therefore deserve special consideration. Their analysis is based on utilisation levels of error-noncumulative periodic tasks (errors of task instances being independent of each other), which can be calculated from execution times and period lengths. Examples for error-noncumulative applications are found in the area of multimedia where tasks receive, process and transmit video, audio, or image data, and in information retrieval applications. Liu et al. [SLC89, SLC92, LLS$^+$91] rate various heuristics for prioritising the optional parts of periodic tasks on multi-processor architectures:

least utilisation: static priorities, suitable for linear error functions

least attained time: dynamic priorities, suitable for convex error functions

first-come-first-serve: dynamic priorities, suitable for concave error functions

shortest period: static priorities, suitable for all error functions

earliest deadline: dynamic priorities, suitable for all error functions

Castorino and Ciccarella [CC00a, Cas96] concentrate on scheduling error-cumulative periodic imprecise computations with hard deadlines for the mandatory and $(1, k)$-firm deadlines for the optional part of the tasks. k is called the cumulative rate of a task. Error-cumulative models are quite common, e.g., in route tracing or real-time control applications for complex industrial plants. The usual approaches to scheduling error-cumulative tasks treat all instances as independent tasks [Leu91, LLS$^+$91] or apply hands-on heuristics [Che92, FL97]. Unfortunately, these solutions are frequently expensive or perform poorly. Castorino and Ciccarella showed it is possible to transform sets of error-cumulative periodic flexible computations with harmonic period lengths and equal cumulative rates to equivalent sets of error-noncumulative tasks in polynomial time, so that the above heuristics become applicable.

Aydın et al. [AMMA99, AMM00, AMA01, AMMA] prove that in fact the performance of lazy slack usage and eager slack usage can be arbitrarily small compared to the optimal scheduler[1]. Hence, the approach suggested is not to decouple the objectives of meeting deadlines for mandatory parts and minimise the error in resource allocation to optional components, in contrary to the models for imprecise computation scheduling described above. Giving up the two-stage approach can significantly increase complexity, but, at the same time, the quality of the resulting schedules. However, in some special cases, practical solutions can be found for the more complex scheme of following both objectives simultaneously. For the important class of independent periodic tasks with non-increasing, differentiable and convex error functions to be scheduled on a uniprocessor system, it can be shown that there are constant optimal service times for each task, such that optimal schedules can always be constructed with every instance of a task allowed the same optimal computation time. With these fixed execution times, the results of classical periodic task scheduling can be applied on imprecise computations.

Deadlines for the mandatory parts of tasks are an integral component of the scheduling algorithms for imprecise computation. We rate this kind of algorithms as belonging to no. 8 of table 9.1.

[1]The terminology is different in this work; in particular, the problem is described in terms of maximising a reward function rather than minimising an error function.

9.5.4 IRIS (Increased Reward with Increased Service) Tasks

The IRIS (increased reward with increased service) model is similar to the imprecise computation approach, but in the IRIS model there is no upper bound to the execution time of tasks. The IRIS task model assumes concave reward functions for all tasks, so that the reward itself increases, but marginal reward (the first derivative of the reward function) decreases with increasing service. However, reward functions need not be convergent, as is the case for anytime algorithms. Probably the direct predecessor to the IRIS task model was the application of so-called approximate processing techniques within real-time schedulers (e.g., by Decker and Lesser [DLW90]).

Dey, Kurose et al. [DKT$^+$93b, DKT93a] investigate the case of dynamically scheduling a set of independent tasks which arrive randomly over time on a single processor. One further assumption is that the release times of tasks are identically distributed. The scheduler is non-anticipative, i.e., it does not take into account any tasks prior to their release time, and it may preempt and resume tasks at any time. Tasks are not composed of smaller components in this basic model of IRIS tasks.

The scheduler works in two phases: first, an optimisation phase running every time a new task arrives in the system determines the optimal service time for each task; second, a low-level scheduling algorithm like EDF determines the execution order on the tasks. The performance metric used is the average reward rate, i.e., the average accrued reward per unit time; upper bounds on the reward rate can be found for special cases, e.g., for task sets with the same reward function for all tasks and arbitrary distributions of release times, and for task sets with arbitrary reward functions and Poisson-distributed release times.

The approach of Dey, Kurose et al. to solve the nonlinear resource allocation problem is as follows: Consider the interval of time starting from the current time and ending with the latest deadline of all tasks currently active. Then partition this interval into a set of disjunctive smaller intervals such that the deadlines of the tasks form the borders of these intervals. The scheduling problem is now equivalent to the problem of allocating service time to the tasks within such intervals so that the average reward rate is maximised. Obviously, two conditions must hold:

- resource constraint: the sum of service allocations to tasks in any interval is lower than or equal to the interval length
- non-negative allocation: avoid negative allocations, because they do not have a physical interpretation

Note that the exact position of the service allocation to tasks is irrelevant for this algorithm, i.e., schedules with the same allocations within each interval are considered equivalent. Both optimal and suboptimal search techniques are described in [DKT$^+$93b].

Çam [Ç00b] extended the idea of IRIS tasks to composite tasks of possibly several mandatory and optional subtasks, thus making is possible to model optional components of tasks, logically group related or dependent tasks. This model generalises both the IRIS task model and the imprecise computation model, as mandatory components need not necessarily be executed prior to optional components of the same task. In fact, the number of possible combinations of components to form a task such that it best suits the available resources is much bigger than in the case of working with two components only. For each task, a family of alternative sets of component tasks is gained by leaving out the least rewarding components. It is assumed that the components of a composite task are released at the same time. Unlike the basic IRIS model by Dey et al., Çam does not use a two-level scheduler, in order to avoid the disadvantages of losing optimum when separating decisions on different objectives. The scheduler scheme suggested by Çam instead can be sketched as follows: Whenever a new composite task arrives, its laxity and processing time are examined. If the laxity is greater than the processing time, the scheduler is invoked. Otherwise, the call to the scheduling algorithm is postponed until there are few tasks waiting for execution (e.g., less than tasks waiting to be scheduled).

Arguing that most schemes for flexible scheduling are too complex to be of practical use for embedded real-time systems, Sugawara and Tatsukawa [ST89] and Liu et al. [Liu88] suggest very simple schemes for sets of periodic tasks with a finite number of service levels each. Small tables of service / quality pairs (segments of discrete performance profiles) are used to store the information needed on the time / quality tradeoff - hence the names *table-based scheduling* (Sugawara and Tatsukawa) and *segmented computation model* (Liu et al.).

The IRIS model belongs to both categories 4 and 8 of table 9.1.

9.5.5 Evaluation and Correlation to this Thesis

Quality functions of our scheduling model can be interpreted as performance profiles of anytime algorithms and flexible computations or reward functions of IRIS tasks. Again, we did not adopt the actual scheduling algorithms from these schemes. Many ideas on how to deal with the cost of scheduling came from the work on deliberation scheduling with anytime tasks, and we use convergent, monotonically increasing quality functions resembling the value functions for flexible computations. Tasks with two components like imprecise computations or bigger sets of mandatory and optional components like in the IRIS model can be achieved in the quality / utility scheduling model through the task hierarchy. Mandatory and optional components can be distinguished by their utility functions and appropriate dependencies between them. However, we do not make explicit use of the semantic differences between these two classes of tasks. Our schedulers never operate in two stages, but solve all problem aspects in the original formulation;

the work described in this section includes both single-stage and two-stage algorithms. A very valuable contribution was the work by Dey, Kurose et al., which provided us with the idea for the class of sliding-window quality / utility schedulers.

9.6 Scheduling with Precedence Constraints

Precedence constraints can arise, e.g., from dataflow dependencies or access to shared resources and are typically specified as a precedence graph. Depending on the interpretation of the nodes and the edges in the graph (e.g., nodes being tasks, jobs, task instances, etc.), the precedence graph may be a tree structure, a directed acyclic graph or a general directed graph. However, many classical scheduling algorithms for real-time systems require tasks to be independent of each other. The usual method of transforming an acyclic precedence graph into a set of independent tasks is by delaying the release times of all tasks until the latest possible finishing time of all predecessors and similarly strengthening the deadline constraints such that they are at most equal to the earliest possible release times of the successors. Transformation of release times takes place in breadth-first manner starting from the root nodes, transformation of the deadlines in reverse breadth-first manner starting from the sinks of the graph. Both components of the technique use the worst-case execution times of tasks to determine the necessary shift in release times and deadlines; the methodology is typically rather pessimistic, i.e., it classifies task sets as non-schedulable that would in effect be very well feasible. Altenbernd backed this argumentation and suggested an alternative scheme which he proved to be less pessimistic [Alt96, AH98].

9.6.1 Precedence Constraints in Timeliness-Flexible Scheduling Schemes

Naturally, scheduling schemes with end-to-end constraints operate with dependency graphs, as this kind of constraints is typically defined on chains or more general directed graphs of tasks.

The BERT scheduler has rudimentary provision of precedence constraints: applications consist of chains of operations called *paths*; no other topology of dependency graphs is allowed. The model of McElhone and Burns uses the concept of release time and deadline transformation to handle precedence constraints and allows to annotate tasks with an *and*/*or* attribute similar to [Gil93]; predecessors of *and* type nodes are interpreted as its components, predecessors of *or* type nodes as alternatives of the successor.

The DASA algorithm (Dependent Applications Scheduling Algorithm) by Clark [Cla90] builds on the work by Locke on best-effort scheduling. It constrains the shape of the allowed time-value functions to binary ones evaluating to a constant positive value before the deadline and to 0 from the deadline onwards. An application is made up of a hierarchy of tasks (the author calls the components at different levels activities, tasks and phases). In this model, Clark was able to define a dynamic scheduler for a set of interdependent preemptive tasks. The precedence graphs in his model must be directed and acyclic; an interesting fact is that these constraints are not known a priori, but arise dynamically at run-time primarily due to the mutual exclusion problem for shared resources; this involves that the direction of a precedence constraint for a task pair connected by a mutual exclusion relationship (i.e., having critical sections on the same resource) may be reversed. On the other hand, of course, some precedence constraints must not change their direction, e.g., those originating from a producer-consumer relationship. Just like Locke's best-effort scheduling algorithm, DASA is most suited for overloaded systems, where there is actually a selection to be made between which tasks to execute and which ones to discard. DASA degrades gracefully under high loads, unlike more simplistic, especially static-priority schemes. The scheduling decisions of DASA are based on value-density, taking into account the estimate of future active tasks whose precedence constraints can be fulfilled to derive a so-called potential value density. Schedules are constructed in a repair-based manner, discarding computations with low potential value density until the schedule becomes feasible.

Zlokapa [Zlo93] follows a similar path, although he claims to present primarily a framework for dynamic scheduling algorithms rather than a specific algorithm. However, the framework he suggest lends itself very easily to handling timeliness-flexible task sets of the best-effort class with precedence constraints. The methodology is based on the perception that many applications can be modelled as a set of task groups rather than one amorphous set of tasks. In hard-real time task models, these task sets are executed as atomic entities; if an abort is necessary after the start of the task group, a rollback has to take place on all computations of the group. On the other hand, non-atomic groups, which can be found in flexible scheduling schemes, do not need this so-called end-to-end scheduling, so a rollback is not normally required in case of task abortion. Based on the perception that many dynamic scheduling algorithms make their decisions either at release time (by means of a schedulability test, resulting in the new task to be either accepted or rejected) or at dispatch time by looking at the prospective gain from each task as late as possible (like in Locke's work), Zlokapa claims that both of these alternatives have undesirable effects: at dispatch time, it might be too late to take alternative actions, while testing schedulability at release time can easily be very pessimistic, as decisions are by nature made in FCFS manner. The goal is to find an optimal point of time when to take the scheduling decisions for each task, the so-called *punctual point*. The methodology called *well-timed scheduling* is compatible with precedence constraints. Part of the scheduling decisions can be made offline: the scheduling algorithm calculates *reflective parameters* for individual tasks by processing their successor tasks, so that at run-time the precedence graph does not have to be processed for each scheduling operation.

9.6.2 Precedence Constraints in Quality-Flexible Scheduling Schemes

The design-to-time model allows to specify so-called nonlocal effects between tasks (rather between a task and a method). Nonlocal effects can affect the duration and/or the quality of the recipient (the target node in terms of a graph structure). An effect is based on the quality of the originating task at the time the recipient begins execution. If a recipient is involved in more than one nonlocal effect, evaluation for all of these takes place at the same time, and no effect requires more than one evaluation of the originating node. Examples for nonlocal effects are:

enables: This kind of nonlocal effect means that the enabling task must have a quality above a threshold or the enabled method will receive zero quality when it is executed.

facilitates: If a task is connected to a method via a *facilitates* effect, then if the facilitating task has nonzero quality, then the facilitated method will have proportionally reduced duration and increased quality.

hinders: Contrary to a *facilitates* effect, a *hinders* effect means that if the hindering task has nonzero quality, then the hindered method will have proportionally increased duration and decreased quality.

In Zilberstein's work on the compilation of anytime algorithms, performance profiles are conditional on the quality of their inputs. This way, dependencies between tasks can be expressed explicitly. However, the evaluation of tasks' qualities influencing their successor nodes takes place only during the compilation procedure. By definition, the contract anytime algorithms which are the result of the compilation cannot be altered or adapted at a later time. More precisely, the scheduler cannot react to changes in task qualities affecting other tasks via dependencies at run-time.

Imprecise computation models either do not take into account precedence constraints at all or employ the simple transformation scheme for release times and deadlines mentioned earlier [CC00a]. The same seems to be true for the IRIS and flexible computations models. One of the rare attempts for scheduling quality-flexible applications with interdependent tasks was made by Hull at al. Starting with work on scheduling linear chains of imprecise computations [HFL96, HFL95], the model finally handles general directed acyclic graphs. At any time, each task has vectors of input and output qualities as well as a vector of resource allocations. Value functions are defined to map a given vector of input qualities and resource allocations (especially processor time) to a vector of output qualities. An example for one-dimensional vectors of input and output qualities and cpu time as the only resource is given in figure 9.8.

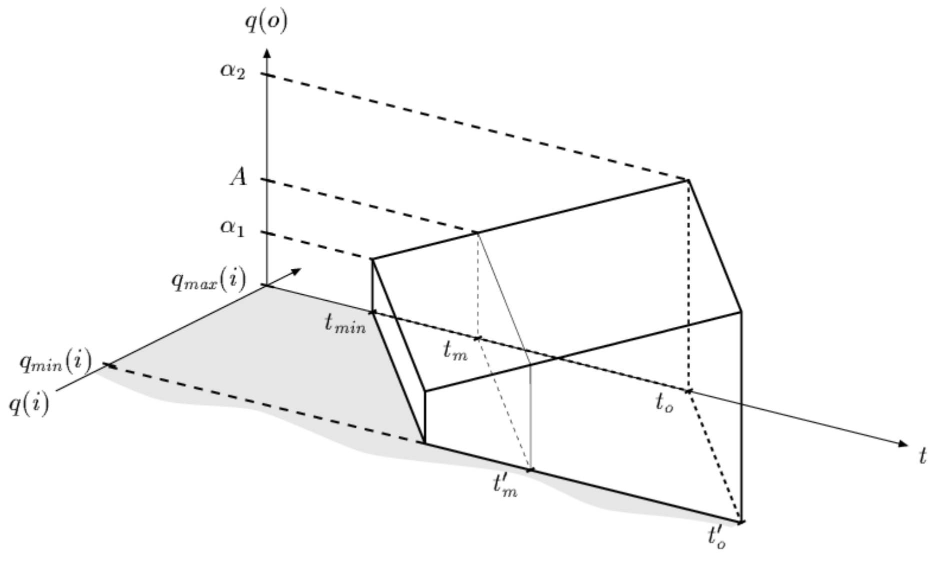


Figure 9.8: Example input quality dependent value function

Information from these functions is used to decide at run-time on an optimal distribution of the resources between the tasks currently waiting for service; the algorithm is myopic, as it does not take into account tasks possibly being released in the future, even if this information might be readily available, e.g., for periodic tasks.

An interesting idea going into the direction of method selection especially for interdependent task sets is the introduction of *and/or* precedence graphs by Gillies [Gil93, GL95]. *And* nodes in this model represent the traditional definition that a task becomes only ready for execution once all of its predecessor nodes have finished. *Or* tasks, however, only require one of their predecessor nodes to finish in order to be able to run. Although this way is becomes much easier for tasks to fulfill the precedence constraints, the scheduling decision itself does not: Gillies could prove that even for the simplest configurations where there is any choice to make (one *or* task with two predecessors, same release times, no deadlines), the problem remains NP-complete. One of the heuristics suggested gradually prunes the precedence graph, leaving *or* type tasks with only the predecessor node representing the shortest path to a root node.

9.6.3 Evaluation and Correlation to this Thesis

We did not opt for guaranteeing a specific order of execution of tasks, like the extension to best-effort scheduling by Zlokapa suggests. Clark's DASA algorithm is also unsuitable, because it relies on relationships between tasks defined via resource access conflicts and we do not model any other resources besides the processor(s). To us, nonlocal effects like in the design-to-time model and the quality-dependent value functions used in the works of Zilberstein and Hull et al. are more promising. We formulated the influence of the value of predecessor tasks on the value of their successor tasks similarly to Hull. Using *and/or* graphs, as Gillies proposed, allows us to use different mechanisms of aggregating value locally for different types of tasks

9.7 Adaptive Scheduling

In our problem class, the scheduler has to adapt its own allocation of processor time dynamically. The following ideas gave incentives to this end.

9.7.1 Imprecise Computations

Feiler and Walker [FW01] suggest an adaptive scheduling mechanism for sets of periodic incremental and design-to-time tasks. As these tasks can be composed of mandatory and optional parts (imprecise computations), the scheduler has to guarantee the execution of some computations before being able to reason about optimising the quality of the others, measured by the deadline miss ratio. At design time, the maximum worst-case execution times of tasks are determined which are allowed while maintaining a feasible schedule. At runtime resources are allocated beyond this statically known worst-case guarantee. The allocation of this additional service time is driven by utility functions defined on the utilisation of tasks. The core of the ability of the system for adaptivity is the dispatch agent. An eligibility list of tasks is kept throughout the runtime of the application, containing tasks in an order indicating their level of improvement that is estimated to be possible when allocating additional resources to them; several policies are described for making such estimates. The two kinds of dispatch agents suggested by the authors are the Incremental Adaptive Dispatch Agent, which communicates with the scheduler independently from the dispatcher for the mandatory computations. On the other hand, the Tunable Adaptive Dispatch Agent communicates with the scheduler only via the mandatory task dispatcher; it is noted that the second alternative usually performs better than the first one, as less context switches are needed and a better control of the implications of optional on mandatory computations reduces the risk of missing deadlines. The feedback data transmitted from the dispatch agent to the scheduler are the actual execution times of tasks and the utilisation of the processors.

9.7.2 Dynamic Window Constrained Scheduling

West [Wes00] investigated feedback control within a scheduling scheme for window-constrained applications called Dynamic Window Constrained Scheduling (DWCS). The primary context for this scheme is communication networks, but the applicability of the methodology to processor scheduling is also demonstrated. The terminology resembles the origins of this work in real-time networking research: packets are transmitted in streams, and loss tolerances describe the maximum number of packets within any fixed-size series of consecutive packets whose transmission may be delayed or disturbed (the window constraint). The deadline miss ratio is the parameter which is monitored and fed back to the scheduler to adapt the dispatch priorities which form the result of the scheduling decisions. A reservation scheme is responsible for guaranteeing a minimum service level offline, and surplus resources are allocated at runtime to maximise overall quality. The core of the system is a modified constant-bandwidth-server scheduling algorithm trying to allocate each of the packet streams sufficient resources (measured in terms of the bandwidth). At runtime, priority in resource allocation is given to the stream with lowest loss-tolerance. Loss-tolerance values of all streams are updated regularly in discrete steps; they are increased if the stream was allocated service time and decreased if this was not possible.

9.7.3 Control-Theoretic Feedback Mechanism

The work of Lu, Stankovic et al. [SLST99, LSTS99, Lu01] presents a control-theoretic approach to feedback scheduling. A general scheduling framework was developed including a PID controller to achieve the ability of a scheduler to adapt itself according to the consequences of prior actions. The first implementation of a feedback policy was derived from the well-known earliest deadline first (EDF) algorithm and called feedback-controlled EDF (FC-EDF) [SLST99, LSTS99]. Later on, also rate-monotonic and deadline monotonic scheduling were implemented within the same framework [Lu01]. Several possible choices of feedback policies (regarding the monitored variable) were investigated, including feedback utilisation control (FC-U) and feedback miss ratio control (FC-M). A PID controller was chosen because it does not require precise knowledge on the dynamics of the system; stability can be guaranteed for first and second order dynamic systems. The advantage of applying a well-known theory instead of an ad-hoc feedback function is that results from very different areas of research could be applied in the work of Lu, Stankovic et al. to derive upper and lower performance bounds, a fact that makes a scheduling algorithm for task sets with unpredictable release and execution times more reliable. Later work [LSA^{+}00, Lu01] employs two PID controllers instead of only one; both the miss ratio and the utilisation are included in the adaptation mechanism at the same time. The new feedback policy is called feedback utilization/miss ratio control (FC-UM), the EDF-based scheduling algorithm FC-EDF^2. Abeni et al. [APB00] build on this work by adding an outer control loop to tackle the known problem of adapting the parameters of the PID controller. Unfortunately, proper settings for the constants of such a controller are frequently outside the intuition of the designer. An additional outer control loop can partly alleviate the problem of finding suitable values, but obviously adds to the complexity and the scheduling overhead.

The advantage of the scheme by Steere at al. [SGG$^+$99] is that only those tasks whose performance requirements are not known a priori are under the control of the feedback mechanism. This can greatly reduce the complexity of computations, depending on the application. The service requirements of other tasks are measured in terms of the repetition rate and the portion of the resources allocated within any such period. Especially the adaptation of the task periods means an additional degree of freedom not normally encountered in similar scheduling models and is limited to certain application scenarios, e.g., in the multimedia, web services or speech recognition areas.

9.7.4 Adaptation of Scheduling Effort

Going further than the work of Dean and Boddy on deliberation scheduling, Horvitz recognised the importance not only of suitably distributing the time spent calculating schedules, but also of finding an optimum balance between the time spent scheduling and the time spent executing these. For this purpose, Horvitz uses decision-theoretic meta-reasoning techniques (meta-reasoning meaning the reasoning on the value of reasoning itself), for which he describes a series of desirable properties, such as the ability of finding close-to-optimal solutions in resource-bounded environments, where the resource bounds apply likewise to the meta-reasoning component and the target components of meta-reasoning.

Horvitz recognised that the cost of deliberation can be directly addressed in the value functions by deriving from the original (so-called object-related) value functions secondary (so-called comprehensive) value functions depending on the time spent preparing for and actually carrying out the scheduling algorithm. Let t_p be the time needed to prepare for the scheduling algorithm, t_b the time spent executing the scheduling algorithm, t_s a relatively small time spent scheduling the tasks of preparation and executing the scheduling algorithm, $V_o : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be the object-related value function and $V_d : \mathbb{R} \to \mathbb{R}$ be a function rating the cost of deliberation. Then the comprehensive value function $V_c : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is defined as

$$V_c(t_p, t_b) = V_o(t_p, t_b) - V_d(t_s + t_p + t_b)$$

Assuming t_s to be constant, the goal is to find optimal values for t_p and t_b such that the comprehensive value is maximal:

$$\max_{t_p,t_b} V_c(t_p, t_b)$$

In many cases, also the parameter t_p can safely be assumed to be constant, so that the problem is reduced to maximising the comprehensive value function by finding an appropriate scheduling effort t_b. Figure 9.9 demonstrates two examples for the calculation of comprehensive value functions from object-related value functions and deliberation value functions [HB90], t_b^* being the optimal value for t_b.

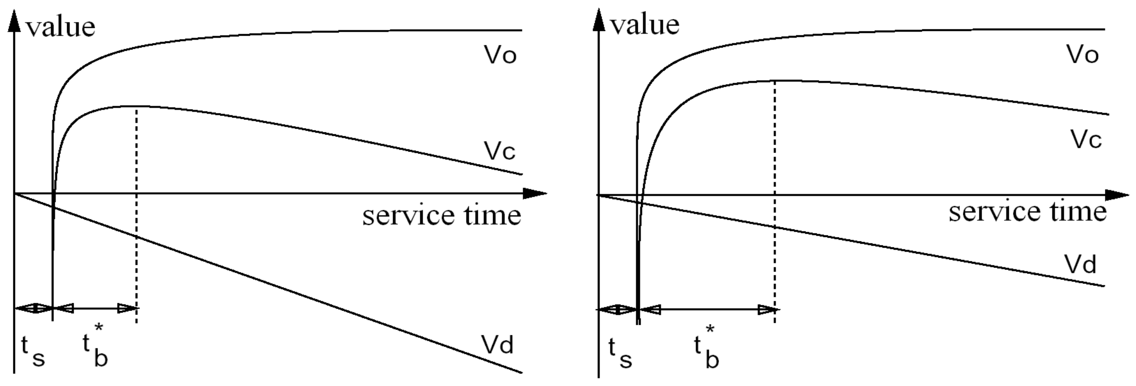


Figure 9.9: Example value functions for flexible computations

9.7.5 Evaluation and Correlation to this Thesis

We cannot give offline guarantees for mandatory parts of tasks as in [FW01], and we do not consider window constraints explicitly in our scheduling algorithms. However, we did adopt some details from the works of Lu, Stankovic at al. and by Horvitz. Lu and Stankovic suggested the control-theoretic approach to adaptivity of a system, and Horvitz recognised the importance of finding a balance between the time spent scheduling and the time spent executing.

Chapter 10

Conclusion

That which separated and distinguished me from others, mattered. That which no one else said or could say, was what I had to say.

André Gide

[L'immoraliste]

In this chapter, we are going to briefly summarise the main topics of this work and assess its possible implications, but also its limitations. We conclude with some ideas for future work in the area of quality / utility scheduling.

10.1 Achievements

In this thesis we presented a general model for applications with flexible timing constraints under the anytime execution paradigm. Both utility functions representing a generalised form of deadlines and quality functions as the performance profiles of anytime algorithms have been used before, but not within one model and for the same task set. Utility and quality functions map time domains into real values; however, the time domains are distinct. As both the timeliness and the quality of computations are expressed by functions of a time domain, quality / utility scheduling lacks both the explicit notion of deadlines and that of given execution times for tasks found in most traditional real-time scheduling schemes. Objective functions have to be defined taking into account these global and task-local time aspects. Instead of concentrating on exactly one objective function, we propose a set of properties we require any prospective objective function to hold. All other parts of the model rely on this set of rather general properties for objective functions only.

Objective functions for the scheduling algorithms are defined such that both an early termination of execution and a longer execution time yield higher values for each task; it is the goal of the scheduler to trade off timeliness for quality of each task. In addition to these conflicting objectives, a second tradeoff is present in dynamic scheduling environments, where the scheduling algorithm shares computational resources with the actual application tasks; both too high and too small a share of processor time reserved for the scheduler yield inferior results. Unlike scheduling schemes using simple inexpensive heuristics, our scheduling algorithms with their possibly complex optimisation and search procedures involved cannot ignore the cost of scheduling. We therefore need to handle the cost of scheduling explicitly in an additional component in the system called a meta scheduler, responsible for distributing processing time between the main scheduler and the application task.

We were able to demonstrate the applicability of our model to existing real-world problems and the feasibility of dynamic scheduling algorithms for the problem class we introduced. The algorithms we proposed are primarily based on local-search optimisation like simulated annealing and decision-theoretic methods like policy iteration for Markov decision processes.

The work includes the development of an integrated specification and simulation environment for scheduling problems and algorithms. Its main components are a graphical editor, graph generators, a time-discrete simulator, visualisation tools, an extensible library of scheduling algorithms and a database-supported benchmarking system.

10.2 System Model

Our work assumes tasks to be arranged in a task/subtask hierarchy with additional *and*/*or* attributes, forming a tree structure to represent an application. In addition to this, value dependencies were introduced as a value-based equivalent to precedence constraints and span a second graph structure on the same task set. The leaves of the task hierarchy tree access the methods of a library of basic algorithms as the most elementary entities of operation.

The model is general enough to allow heterogeneous multiprocessor systems as the target architecture. Tasks are preemptive, and we do not consider context switch costs. However, we do prohibit migration of tasks between processors, because it seems too unrealistic to assume zero cost for this in fact very expensive procedure. Tasks are generally divided into separate units of operation with exactly the same implementation in terms of the underlying methods or subtask structure. These units are invoked in strictly determined order and are called instances of the task. The schedulers do not need to know exact release times of task instances (in which case, scheduling could be done offline). However, we assume a stochastic distribution for the release times to be known. In many cases, these release times are (approximately) equidistant; such periodic tasks have been investigated along with other distributions of release times for tasks.

10.3 Interpretation of Experimental Data

Experimental results show that the scheduling algorithms we developed are primarily overload methods. At utilisation levels below 1, simple heuristics like utility-density scheduling[1] can achieve the same value for schedules with much smaller effort. Only if there is potential to make decisions not only when to execute tasks, but also which ones to execute and which ones to discard or on their overall execution time, the more complex algorithms of this thesis show their benefits.

Simulation also demonstrates that the performance of the scheduling algorithms degrades gracefully in overload, especially if the scheduling window is chosen big enough, so that a good estimate of the set of ready task instances in the near future is available at any time. If the scheduling window is too small, the scheduler has very little data on the long-term benefit of allocating processor time to individual tasks. Remember that both possible quality increases and utility decreases beyond the end of the scheduling window are not taken into account for scheduling decisions. However, it has to be noted that at higher utilisation levels the scheduler needs a considerably longer computation time to find an optimal solution, which can, of course, be explained by the larger search space. In any case, schedule qualities converge for rising effort.

Another important parameter influencing the performance of the schedulers is the accuracy of the release time estimate and hence the stochastic distribution of release times. If the variance of release time distributions is small, partial schedules calculated for the task set of the near future need hardly any corrections during execution. On the other hand, if the estimates are not very reliable, schedule adaptations and rescheduling are frequently necessary; with rising variance, the choice between decreasing schedule quality and increasing scheduling effort becomes more difficult. In our model for the release types of tasks, a big variance results from a large maximum release jitter of periodic tasks and small release probabilities of aperiodic tasks. We also found that the local-search class of our schedulers is in general more susceptible to the influence of the uncertainty in the release times than the decision-theoretic scheduler, which calculates alternative actions for less likely situations in advance.

[1]a generalisation of the well-known EDF scheduler prioritising tasks with steepest decrease of utility

Finally, in our experiments we could show that the PID controller is able to stabilise and find appropriate values for the scheduling allowance within a small number of scheduling phases. The flexible scheme for deciding on the scheduling allowance usually outperforms a fixed reservation of processor time for the scheduler.

10.4 Potentials and Limitations

Quality / utility scheduling can be applied to a wide range of existing real-world scenarios. It can hence serve as the basis of comparison between similar problems and the specific solutions devised for them. Scheduling algorithms for the general quality / utility problem can be applied to all classes of problems which can be subsumed under the more general model.

Hard deadlines can easily be represented by utility functions. However, it depends on the performance of the optimisation algorithm whether these can be met. Similarly, execution orders contradicting precedence constraints are not forbidden in our model, but only penalised by the objective function. We can therefore not guarantee any specific order of tasks; however, wrong execution orders, just like missed deadlines, can be assigned large or even infinite penalties. Many approaches to flexible scheduling can be expressed within the quality / utility model, among them end-to-end constraints, anytime algorithms, best-effort scheduling and others. Others do not fit into our framework, e.g., window constraints.

Both the objective functions and the scheduling algorithms we suggested rely on discrete local and global times and do not easily extend to continuous time domains. The quality / utility scheduling model as presented here is therefore not applicable to continuous-time problems, even though we sometimes use continuous definitions of quality and utility functions, implicitly assuming discretisation whenever necessary.

10.5 Open Problems and Future Work

The assumption in this thesis was that context switch costs are small and can be ignored. Furthermore, migration of tasks, i.e., continuing computation of a task on a different processor from the one it had been running on before preemption, was disallowed altogether. Finally, communication costs for tasks with dataflow dependencies executed on different processors were not taken into account. However, with a suitable cost model, these restrictions can be lifted. Context switch costs depend on the amount of local data for each task; these usually include register contents, the local heap, etc. (the task control block). As a first attempt, context switch costs could be assumed constant for each processor; a more sophisticated model would allow to parameterise this specification to accommodate task-specific memory requirements. Context switch costs should be modelled together with general communication costs; the difference between the cost models for task migration and inter-task communication is merely in the granularity of the task set derived from an application. In general, communication costs are more difficult to assess than context switch costs. They depend on factors like the topology of a multiprocessor or distributed system, network bandwidth, or bus capacity.

Another direction of research are possible simplifications of the original problem setting for which significantly more efficient solutions exist. If we look at one of the original objective functions on a task set $\mathbb{T}'$

$$\sum_{T\in\mathbb{T}'} \max_{t'\leq t} u_T(t') \cdot q_T(\tau_T(t')),$$

we see that complexity problems arise from the potential for resource allocation to a task at any time after its release and the maximum operator requiring the objective function to be evaluated at any point in time. For these reasons, the value of a schedule depends on the exact position of resource allocations on the timeline. As an alternative, the objective function

$$\sum_{T\in\mathbb{T}'} u_T(s_T) \cdot q_T(\tau_T(s_T))$$

with s_T being the stopping time of task T simplifies the search for an optimal schedule, because the objective function needs to be evaluated only once, namely at its stopping time; no further computing time can be allocated after that. Once we decide on an order for the release and stopping times of tasks and the execution time for all tasks, we know an optimal schedule exists which at any time executes the ready task with the earliest stopping time. Appropriate orders of stopping and execution times of tasks can be found by local search, just as in the schedulers we described in this thesis; however, the search space is much smaller for the modified problem. It can be hoped that real-world applications can be linked to this modified problem class.

An important issue of future work will be the analysis of the PID controller and its parameters. Our hope is to gain analytical results to be able to make a better informed choice of parameters. It remains to be seen whether a PID or a PI controller is best suited for the quality / utility meta scheduler.

Obviously, the applicability of quality / utility scheduling to real-world problems should be demonstrated not only by modelling and simulation. The deployment of specialised schedulers to real quality and timeliness flexible applications is a step yet to be taken.

Bibliography

[AB99] S.A. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, 1999.

[AH98] P. Altenbernd and H. Hansson. The slack method: A new method for static allocation of hard real-time tasks. *Real-Time Systems*, 15:103–130, 1998.

[Alt96] P. Altenbernd. *Timing Analysis, Scheduling and Allocation of Periodic Hard Real-Time Tasks*. PhD thesis, Fakultät für Mathematik und Informatik, Universität-GH Paderborn, 1996.

[AMA01] H. Aydın, R. Melhem, and P.M. Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 50(2):111–130, February 2001.

[AMM00] H. Aydın, R. Melhem, and D. Mossé. Tolerating faults while maximizing reward. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, 2000.

[AMMA] P.M. Alvarez, R. Melhem, D. Mossé, and H. Aydın. An incremental server for scheduling overloaded real-time systems. to appear in: IEEE Transactions on Computers.

[AMMA99] H. Aydın, R. Melhem, D. Mossé, and P.M. Alvarez. Optimal reward-based scheduling of periodic real-time tasks. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[APB00] L. Abeni, L. Palopoli, and G. Buttazzo. On adaptive control techniques in real-time resource allocation. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, 2000.

[BA02] G. Buttazzo and L. Abeni. Smooth rate adaption through impedance control. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, 2002.

[BB97] G. Bernat and A. Burns. Combining (n m) hard deadlines and dual priority scheduling. In *Proceedings of the IEEE Symposium on Real-Time Systems*, December 1997.

[BB01] G. Bernat and A. Burns. Three obstacles to flexible scheduling. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, 2001.

[BB02] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems Journal*, 22(1/2):49–75, 2002.

[BBB02] I. Broster, G. Bernat, and A. Burns. Weakly hard constraints on controller area network. In *Proceedings of the 14th Euromicro Conference on Real-time Systems*, 2002.

[BBL01] G. Bernat, A. Burns, and A. Llamosí. Weakly-hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, 2001.

[BD89] M. Boddy and T. Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[BD94] M. Boddy and T. Dean. Decision-theoretic deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–286, 1994.

[Ber98] G. Bernat. *Specification and analysis of weakly hard real-time systems*. PhD thesis, Departament de Ciències Matemàtiques i Informàtica, Universitat de les Illes Balears, 1998.

[BH90] P.P. Bonissone and P.C. Halverson. Time-constrained reasoning under uncertainty. *The Journal of Real-Time Systems*, 2(1/2):25–45, 1990.

[Bod91] M. Boddy. *Solving time-dependent problems: A decision-theoretic approach to planning in dynamic environments*. PhD thesis, Department of Computer Science, Brown University, Providence, RI, 1991.

[BP00] A. Bavier and L. Peterson. The power of virtual time for multimedia scheduling. In *Proceedings of NOSSDAV 2000*, June 2000.

[BPB$^+$00] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46:305–325, 2000.

[BPM99] A. Bavier, L. Peterson, and D. Mosberger. BERT: A scheduler for best effort and realtime tasks. Technical Report TR-602-99, Princeton University, March 1999.

[BRC02] P. Balbastre, I. Ripoll, and A. Crespo. Schedulability analysis of window-constrained execution time tasks for real-time control. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, 2002.

[Bus04] B. Busse. Implementierung von Scheduling-Algorithmen in Pascha. Technical report, Lehrstuhl für Rechnerstrukturen, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[Cas96] A. Castorino. *Architetture hardware e software per sistemi di elaborazione real-time*. PhD thesis, Università di Roma ’La Sapienza’, 1996.

[CC00a] A. Castorino and G. Ciccarella. Algorithms for real-time scheduling of error-cumulative tasks based on the imprecise computation approach. *Journal of Systems Architecture*, 46:587–600, 2000.

[Ç00b] H. Çam. An on-line scheduling policy for IRIS real-time composite tasks. *The Journal of Systems and Software*, 52:25–32, 2000.

[Che92] I.K. Cheong. *Scheduling imprecise hard real-time jobs with cumulative error*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Che02] A.M.K. Cheng. *Real-Time Systems*. John Wiley & Sons, 2002.

[Cla90] R.K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie-Mellon University, 1990.

[CM96] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value functions. *Real-Time Systems*, 10:293–312, 1996.

[D’A89] B. D’Ambrosio. Resource bounded agents in an uncertain world. In *Proceedings of the Workshop on Real-Time Artificial Intelligence (IJCAI-89)*, 1989.

[DB88] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[DC99] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[Dem02] S. Demirci. Praktikumsbericht zum DASA/ND Algorithmus (DASA/ No Dependency). Technical report, Lehrstuhl für Rechnerstrukturen, Fakultät für Mathematik und Informatik, Universität Passau, 2002.

[DK96] T. Dean and S. Kambhampati. Planning and scheduling. In *Handbook of Computer Science and Engineering*. CRC Press, 1996.

[DKKN95] T. Dean, L.P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.

[DKT93a] J.K. Dey, J.F. Kurose, and D. Towsley. Efficient on-line processor scheduling for a class of IRIS (inreasing reward with increasing service) real-time tasks. Technical Report 93-09, Department of Computer Science, University of Massachusetts, Amherst, January 1993.

[DKT$^+$93b] J.K. Dey, J.F. Kurose, D. Towsley, C.M. Krishna, and M. Girkar. Efficient on-line processor scheduling for a class of IRIS (inreasing reward with increasing service) real-time tasks. In *Proceedings of 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 217–228, 1993 1993.

[DLW90] K. Decker, V. Lesser, and R. Whitehair. Extending a blackboard architecture for approximate processing. *The Journal of Real-Time Systems*, 2(1/2), January 1990.

[EG91] E.J.Horvitz and G.Rutledge. Time-dependent utility and action under uncertainty. In *Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence*, pages 151–158, 1991.

[Ehr02] K. Ehrnböck. Strukturierung und Implementierung eines Referenzmodells für Echtzeitscheduling. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2002.

[FCS$^+$02] K. Fritzsche, A. Can, H. Shen, C. Tsai, J. Turner, H.L. Tanenbaum, C.V. Stewart, and B. Roysam. Automated model based segmentation, tracing and analysis of retinal vasculature from digital fundus images. In *Angiography and Plaque Imaging: Advanced Segmentation Techniques*. CRC Press, 2002.

[FL97] W.-C. Feng and J.W.S. Liu. Algorithms for scheduling real-time tasks with input error and end-to-end deadlines. *IEEE Transactions on Software Engineering*, 23(2), February 1997.

[Fli04] J. Flierl. Entwurf und Implementierung einer datenbankgestützten Benchmark-Umgebung für Echtzeit-Schedulingalgorithmen. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[FW01] P.H. Feiler and J. Walker. Adaptive feedback scheduling of incremental and design-to-time tasks. In *Proceedings of 23rd International Conference on Software Engineering (ICSE 2001)*, 2001.

[Gar96] A.J. Garvey. *Design-to-Time Real-Time Scheduling*. PhD thesis, University of Massachusetts, Amherst, 1996.

[GHWB95] A. Goforth, N.R. Howes, J.D. Wood, and M.J. Barnes. Real-time design with peer tasks. Technical Report 110367, National Aeronautics and Space Administration (NASA), October 1995.

[Gil93] D.W. Gillies. *Algorithms to schedule tasks with AND/OR precedence constraints*. PhD thesis, Department of Computer Science, University of Illinois, 1993.

[GKS95] M. Gergeleit, J. Kaiser, and H. Streich. Checking timing constraints in distributed object-oriented programs. In *Proceedings of the Workshop on Object-Oriented Real-Time Systems at the IEEE Symposium on Parallel and Distributed Processing*, 1995.

[GL93] A.J. Garvey and V. Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1491–1502, 1993.

[GL95] D.W. Gillies and J.W.S. Liu. Scheduling tasks with and/or precedence constraints. *SIAM Journal of Computing*, 24(4):797–810, 1995.

[GL96a] A. Garvey and V. Lesser. Design-to-time scheduling and anytime algorithms. *SIGART Bulletin*, 7(3), January 1996.

[GL96b] A. Garvey and V. Lesser. Issues in design-to-time real-time scheduling. In *AAAI Fall 1996 Symposium on Flexible Computation*, November 1996.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computing & Operations Research*, 13:533–549, 1986.

[GTdW93] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[HAR99] B. Hamidzadeh, Y. Atif, and K. Ramamritham. To schedule or to execute: Decision support and performance implications. *The International Journal of Time-Critical Computing Systems*, 16:281–313, 1999.

[HB90] E.J. Horvitz and J.S. Breese. Ideal partition of resources for metareasoning. Technical Report KSL-90-26, Stanford Knowledge Systems Laboratory, 1990.

[HFL95] D. Hull, W.-C. Feng, and J.W.S. Liu. Enhancing the performance and dependability of real-time systems. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, 1995.

[HFL96] D. Hull, W.-C. Feng, and J.W.S. Liu. Operating system support for imprecise scheduling. In *Proceedings of the AAAI Fall Symposium on Flexible Computation*, 1996.

[HKGL03] M. Hadad, S. Kraus, Y. Gal, and R. Lin. Temporal reasoning for a collaborative planning agent in a dynamic environment. *Annals of Mathematics and Artificial Intelligence*, 37(1):331–380, 2003.

[HKWF03] J. Huang, Ch. Krasic, J. Walpole, and W.-C. Feng. Adaptive live video streaming by priority drop. In *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance*, July 2003.

[Hor87] E.J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*, 1987.

[Hor88] E.J. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988.

[HR95] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, 44(12), December 1995.

[JLT85] E.D. Jensen, C.D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1985.

[Jün01] Ch. Jünger. Entwurf und Implementierung einer grafischen Benutzeroberfläche eines Simulationstools für Echtzeit-Scheduling. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2001.

[KJV83] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kor85] R.E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 1(27):97–109, 1985.

[Kor87] R.E. Korf. Real-time heuristic search: First results. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 133–138, 1987.

[Kor88] R.E. Korf. Real-time heuristic search: New results. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 139–144, 1988.

[Kor90] R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 3(42):189–212, 1990.

[Leu91] J.Y.T. Leung. Research in real-time scheduling. In M. van Tilborg and G.M. Koob, editors, *Foundation of Real-Time Computing: Scheduling and Resource Management*, pages 31–62. Kluwer Academic Publishers, 1991.

[Lim04] K. Limpöck. Implementierung der Utility-Function und Korrektur der Qualitätsberechnung in Pascha. Technical report, Lehrstuhl für Rechnerstrukturen, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[Liu88] J. Liu. Timing constraints and algorithms. In *Report on the Embedded AI Languages Workshop*, 1988.

[Liu00] J. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 30:46–61, January 1973.

[LLS$^+$91] J.W.S. Liu, K.J. Lin, W.K. Shih, A.C. Yu, J. Y. Chung, and W. Zhao. Algorithms for imprecise computations. *IEEE Computer*, 24:58–68, 1991.

[Loc86] C.D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, 1986.

[LSA$^+$00] Ch. Lu, J.A. Stankovic, T.F. Abdelzaher, G. Tao, S.H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2000.

[LSR$^+$02] G. Lin, C.V. Stewart, B. Roysam, K. Fritzsche, and H.L. Tanenbaum. Predictive scheduling algorithms for real-time feature extraction and spatial referencing: Application to retinal image sequences. IEEE Transactions on Biomedical Engineering, October 2002.

[LSTS99] Ch. Lu, J.A. Stankovic, G. Tao, and S.H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1999.

[Lu01] Ch. Lu. *Feedback Control Real-Time Scheduling*. PhD thesis, University of Virginia, 2001.

[Luc04] D. Lucic. Implementierung der Utility-Function und Korrektur der Qualitätsberechnung. Technical report, Lehrstuhl für Rechnerstrukturen, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[LZ98] H. Liu and M.E. Zarki. Adaptive source rate control for real-time wireless video transmission. *Mobile Networks and Applications*, (3):49–60, 1998.

[MB00] Ch. McElhone and A. Burns. Scheduling optional computations for adaptive real-time systems. *Journal of Systems Architecture*, 46:49–77, 2000.

[MC92] P. Muhlethaler and K. Chen. Generalized scheduling on a single processor in real-time systems based on time value functions. Technical Report 1759, Unité de Recherche INRIA-Roquencourt, Institut National de Recherche en Informatique et en Automatique, 1992.

[MMM00] A. Mittal, G. Manimaram, and C. Siva Ram Murthy. Integrated dynamic scheduling of hard and QoS degradable real-time tasks in multiprocessor systems. *Journal of Systems Architecture*, 46:793–807, 2000.

[MP93] T.E. Morton and D.W. Pentico. *Heuristic Scheduling Systems*. John Wiley, 1993.

[MPR99] D. Mossé, M.E. Pollack, and Y. Ronén. Value-density algorithms to handle transient overloads in scheduling. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, 1999.

[MRR$^+$53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, (21):1087–1092, 1953.

[Mül04] Ch. Müller. Optimierunng der Zustandsverwaltung und der Visualisierungskomponenten in Pascha. Technical report, Lehrstuhl für Rechnerstrukturen, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[NPB02] A. Nakao, L. Peterson, and A. Bavier. Constructing end-to-end paths for playing media objects. *Computer Networks*, 3(38):373–389, February 2002.

[Ram97] P. Ramanathan. Graceful degradation in real-time control applications using (m,k)-firm guarantee. In *Proceedings of the FTCS*, pages 132–141, 1997.

[Ram99] P. Ramanathan. Overload management in real-time control application using (m,k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[Reg01] J.D. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[Rej99] R. Rejaie. *An End-to-End Architecture for Quality Adaptive Streaming Applications in the Internet*. PhD thesis, Computer Science Department, University of Southern California, 1999.

[RHE99] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *Proceedings of the IEEE Infocom'99*, 1999.

[RKJZ99] A. Raha, S. Kamat, X. Jia, and W. Zhao. Using traffic regulation to meet end-to-end deadlines in ATM networks. *IEEE Transactions on Computers*, 48(9), September 1999.

[RZ91] S.J. Russel and S. Zilberstein. Composing real-time systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 212–217, 1991.

[Sau99] J. Sauer. Knowledge-based systems techniques and applications in scheduling. In T.L. Leondes, editor, *Knowledge-Based Systems Techniques and Applications*. Academic Press, 1999.

[Sau03] J. Sauer. Planning and scheduling - an overview. In L. Hotz and T. Krebs, editors, *Proceedings des Workshops zur KI 2003*, pages 158–161, 2003.

[Sch02] R. Schiller. Statistikvisualisierung in Pascha. Technical report, Lehrstuhl für Rechnerstrukturen, Fakultät für Mathematik und Informatik, Universität Passau, 2002.

[Sch03a] T. Schwarzfischer. Application of simulated annealing to anytime scheduling problems with additional timing constraints. In *Proceedings of the 5th Metaheuristics International Conference (MIC2003)*, August 2003.

[Sch03b] T. Schwarzfischer. A novel model for the budget sharing problem in a real-time environment. In *Proceedings of the 2003 International Conference on Artificial Intelligence (IC-AI)*, June 2003.

[Sch03c] T. Schwarzfischer. Quality and utility - towards a generalization of deadline and anytime scheduling. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, June 2003.

[Sch03d] T. Schwarzfischer. Using value dependencies to schedule complex soft-real-time applications with precedence constraints. In *Proceedings of the 1st Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA)*, August 2003.

[Sch04a] R. Schiller. Entwicklung und Untersuchung von adaptiven Verfahren zum Deliberation Scheduling für eine wertebasierte und dynamische Problemstellung. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[Sch04b] T. Schwarzfischer. Closed-loop online scheduling with timing constraints and quality profiles on multiprocessor architectures. In *Proceedings of the Workshop on Integrating Planning into Scheduling at the 14th International Conference on Automated Planning and Scheduling (ICAPS2004)*, June 2004.

[SGG$^+$99] D.C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[SLC89] W.K. Shih, W.S. Liu, and J.Y. Chung. Fast algorithms for scheduling imprecise computations. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 12–19, 1989.

[SLC92] W.K. Shih, W.S. Liu, and J.Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal of Computing*, 7 1992.

[SLST99] J.A. Stankovic, Ch. Lu, S.H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 11–20, 1999.

[SRS$^+$01] H. Shen, B. Roysam, C.V. Stewart, J.N. Turner, and H.L. Tanenbaum. Optimal scheduling of tracing computations for real-time vascular landmark extraction from retinal fundus images. *IEEE Transactions on Information Technology for Biomedicine*, 5:1:77–91, 2001.

[SSR$^+$03] H. Shen, C.V. Stewart, B. Roysam, G. Lin, and H.L. Tanenbaum. Frame-rate spatial referencing based on invariant indexing and alignment with application to on-line retinal image registration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25:3, 2003.

[ST89] T. Sugawara and K. Tatsukawa. Table-based QoS control for embedded real-time systems. *ACM SIGPLAN Notices*, 34(7), 7 1989.

[Str94] H. Streich. TaskPair-Scheduling: An approach for dynamic real-time systems. In *Proceedings of the 2nd Workshop on Parallel and Distributed Real-Time Systems*, 1994.

[TWW87] H. Tokuda, J.W. Wendorf, and H.J. Wang. Implementation of a time-driven scheduler for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 271–280, 1987.

[VR99] R.D. Venkataramana and N. Ranganathan. Multiple cost optimization for task assignment in heterogeneous computing systems using learning automata. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, 1999.

[Wei04] M. Weindler. Entwurf und Implementierung von Generatoren für kombinierte Task- und Ressourcengraphen. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[Wen88] J.W. Wendorf. Implementation and evaluation of a time-driven scheduler for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 172–180, 1988.

[Wes00] R. West. *Adaptive Real-Time Management of Communication and Computation Resources*. PhD thesis, Georgia Institute of Technology, 2000.

[WHZ$^+$01] D. Wu, Y.T. Hou, W. Zhu, Y.-Q. Zhang, and J.M. Peha. Streaming video over the internet: Approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3), March 2001.

[WL00] T. Wagner and V. Lesser. Design-to-criteria scheduling: Real-time agent control. In *Proceedings of AAAI 2000 Spring Symposium on Real-Time Autonomous Systems*, pages 89–96, March 2000.

[Zac04] P. Zach. Entwicklung und Implementierung eines Verfahrens zum dynamischen Scheduling von Anytime-Tasks zur Qualitäts- und Nutzenoptimierung auf der Grundlage Markovscher Entscheidungsprozesse. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[Zha91] L. Zhang. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.

[Zil93] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, University of California at Berkeley, 1993.

[Zim04] R. Zimmermann. Implementierung eines Client/Server-Systems zur Anbindung von PASCHA an CPLEX. Technical report, Lehrstuhl für Rechnerstrukturen, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[Zlo93] G. Zlokapa. *Real-Time Systems: Well-Timed Scheduling and Scheduling with Precedence Constraints*. PhD thesis, University of Massachusetts, 1993.

[ZR87] W. Zhao and K. Ramamritham. Simple and integrated heuristic algorithm for scheduling tasks with time and resource constraints. *Journal of System and Software*, 7:195–205, 1987.

[ZRS87a] W. Zhao, K. Ramamritham, and J. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):pp 949–960, August 1987.

[ZRS87b] W. Zhao, K. Ramamritham, and J. Stankovic. Scheduling tasks with resource requirements in hard real–time systems. *IEEE Transactions on Software Engineering*, SE-13(5):564–677, May 1987.

Appendix A

List of Symbols

Symbol	**Type**	**Context**	**Description**	**Page**
$\#rep$	$\in \mathbb{N}_0$	simulated annealing	number of search steps within a temperature level	53
$\alpha_{P,I}$	$\in (\mathbb{LT}_{P,I})^{\mathbb{LTF}_{P,I} \times \mathbb{GT}}$	extended model	allocation function of method instance I with regard to processor P	73
$\alpha_{P,I}$	$\in (\mathbb{LT}_{P,I})^{\mathbb{LTF}_{P,I} \times \mathbb{GT}}$	extended model	local allocation function for task instance I with regard to processor P	84
α_T	$\in (\mathbb{LT}_T)^{\mathbb{LTF}_T \times \mathbb{GT}}$	basic model	allocation function	21
α_T	$\in (\mathbb{LT}_T)^{\mathbb{LTF}_T \times \mathbb{J}^{\geq \vartheta}_{T,[t_0;t_0+ws[,\mathbb{T}'}}$	reactive scheduling	allocation function in terms of elementary intervals	48
Δ_i	$\in \mathbb{N}$	meta scheduling	size of the i-th partial schedule	122
Δ_{min}	$\in \mathbb{N}$	meta scheduling	user-defined minimum partial schedule length	123
$\Delta slope_i$	$\in \mathbb{R}_0^+$	meta scheduling	change in value density between two consecutive scheduling phases	123
$\delta_T^{\vartheta_u}$	$\in \mathbb{GT} \cup \{\infty\}$	results	threshold deadline	169
$\eta_M^{\vartheta_q}$	$\in \bigcup_{k \in \mathbb{N}_0} \mathbb{LT}_{P,I_M^k} \cup \{\infty\}$	results	threshold execution time	168
ϑ_q	$\in \mathbb{R}_0^+$	results	quality threshold	168

Symbol	**Type**	**Context**	**Description**	**Page**
ϑ_u	$\in \mathbb{R}_0^+$	reactive scheduling	utility threshold	45, 169
λ	$\in \mathbb{R}$	Lagrange optimisation	Lagrange multiplier	61
ν	$\in \mathbb{N}_0$	local search	number of ready tasks	58
π	$\in \hat{A}^{\hat{\mathbb{S}} \cup \{s_0, s_{out}, s_{err}\}}$	MDPs	policy	65
$\sigma_{I,\vec{\tau}}$	$\in \mathbb{GT}$	extended model	earliest time of processor allocation to instance I	107
$\vec{\tau}$	$\in \mathbb{LTF}_{\mathbb{I}'}$	extended model	vector of local time functions for method instances in $\mathbb{I}'$	73
$\vec{\tau}$	$\in \mathbb{LTF}_{\mathbb{I}'}$	extended model	vector of local time functions for instances in $\mathbb{I}$	84
$\tau_{P,I}$	$\in (\mathbb{LT}_{P,I})^{\mathbb{GT}}$	extended model	local time function of method instance I with regard to processor P	73
$\tau_{P,I}$	$\in (\mathbb{LT}_{P,I})^{\mathbb{GT}}$	extended model	local time function for task instance I with regard to processor P	84
τ_T	$\in \mathbb{LTF}_T$	basic model	local time function	21
τ_T	$\in (\mathbb{LT}_T)^{\mathbb{J}^{\geq\vartheta}_{T,[t_0;t_0+ws[,\mathbb{T}'}}$	reactive scheduling	local time function in terms of elementary intervals	48
$\underline{\tau}_T$	$\in (\mathbb{GT} \cup \{\infty\})^{\mathbb{LT}_T}$	basic model	local timeliness function	21
φ_T	$\in \mathbb{N}_0$	extended model	phase shift of periodic or aperiodic task T	76 / 77
ψ	$\in \mathbb{N}_0$	local search	number of time units to distribute	58
$\mathbb{A}$	$\equiv \mathbb{T}'$	MDPs	set of possible actions	64
$\hat{\mathbb{A}}$	$\subseteq \mathbb{A}$	MDPs	action envelope	64
a	$\in (2^{\mathbb{T}})^{\mathbb{T} \cup \mathbb{M}}$	extended model	ancestor relation in the task hierarchy	81

Symbol	**Type**	**Context**	**Description**	**Page**
$a_{t'}$	$\in \mathbb{T}(t)$	MDPs	task running at time $t' \leq t$	63
c	$\in (2^{\mathbb{M}})^{\mathbb{T}}$	extended model	child function for tasks	74
c	$\in (2^{\mathbb{I}_{\mathbb{M}}})^{\mathbb{I}_{\mathbb{T}}}$	extended model	child function for task instances	75
c	$\in (2^{\mathbb{T} \cup \mathbb{M}})^{\mathbb{T}}$	extended model	child function in the task hierarchy	80
c	$\in (2^{\mathbb{I}_{\mathbb{T}} \cup \mathbb{I}_{\mathbb{M}}})^{\mathbb{I}_{\mathbb{T}}}$	extended model	child function in the instance hierarchy	80
C_d	$\in \mathbb{R}$	meta scheduling	constant factor for derivative component of PID controller	126
cF	$\in]0;1[$	simulated annealing	cool-down factor	54
C_i	$\in \mathbb{R}$	meta scheduling	constant factor for integral component of PID controller	126
C_p	$\in \mathbb{R}$	meta scheduling	constant factor for proportional component of PID controller	126
$delay$	$\in (\mathbb{N}_0)^{\mathbb{T}_s \times \mathbb{T}_s}$	extended model	delay in distance numbers on dependency edges	101
err_i	$\in \mathbb{R}$	meta scheduling	error function for PID controller	125
f	$\in \mathbb{R}^{\mathbb{R}^n}$	Lagrange optimisation	objective function for Lagrange multiplier optimisation	61
$\mathbb{GT}$	$\equiv \mathbb{N}_0$	basic model	global time	17
g	$\in \mathbb{R}^{\mathbb{R}^n}$	Lagrange optimisation	constraint function for Lagrange multiplier optimisation	61
$\mathbb{I}_{\mathbb{M}'}$	$\equiv \bigcup\limits_{M \in \mathbb{M}'} \mathbb{I}_M$	extended model	set of all instances of methods $M \in \mathbb{M}'$	72
$\mathbb{I}_{\mathbb{T}'}$	$\equiv \bigcup\limits_{T \in \mathbb{T}'} \mathbb{I}_T$	extended model	set of all instances of tasks in $\mathbb{T}'$	74

Symbol	Type	Context	Description	Page
$\mathbb{I}_M$	(enumeration)	extended model	set of all instances of method M	72
$\mathbb{I}_T$	(enumeration)	extended model	set of all instances of task T	74, 82
I_M^k	$\in \mathbb{I}_M$	extended model	k-th instance of method M	72
I_T^k	$\in \mathbb{I}_T$	extended model	k-th instance of task T	74
iat_T	$\in \mathbb{N}_0$	extended model	minimum interarrival time of aperiodic task T	77
$\overline{iat}_T$	$\in \mathbb{N}_0 \cup \{\infty\}$	results	mean interarrival time of task T	169
id_T	$\in \mathbb{R}_0^+$	results	instance density of task T	171
$impTh$	$\in \mathbb{R}$	tabu search	threshold for classifying the relative improvement	58
$\mathbb{J}_{\mathbb{T}'}$	$\subseteq 2^{\mathbb{GT}}$	reactive scheduling	set of elementary intervals for task set $\mathbb{T}'$	44
$\mathbb{J}_{\mathbb{T}',[t_0;t_0+ws[}^{\geq\vartheta_u}$	$\subseteq 2^{\mathbb{GT}}$	reactive scheduling	set of elementary intervals for tasks in $\mathbb{T}'$ within scheduling window $[t_0;t_0+ws[$ above utility threshold ϑ_u	45
$\mathbb{J}_T$	$\subseteq 2^{\mathbb{GT}}$	reactive scheduling	set of elementary intervals for task T	43
$\mathbb{J}_{T,[t_0;t_0+ws[}$	$\subseteq 2^{\mathbb{GT}}$	reactive scheduling	set of elementary intervals for task T within scheduling window $[t_0;t_0+ws[$	44
$\mathbb{J}_{T,[t_0;t_0+ws[}^{\geq\vartheta_u}$	$\subseteq 2^{\mathbb{GT}}$	reactive scheduling	set of elementary intervals for task T within scheduling window $[t_0;t_0+ws[$ above utility threshold ϑ_u	45
$\mathbb{J}_{T,\mathbb{T}'}^{\geq\vartheta_u}$	$\subseteq 2^{\mathbb{GT}}$	reactive scheduling	set of elementary intervals for task T at granularity of task set $\mathbb{T}'$ above utility threshold ϑ_u	46

Symbol	**Type**	**Context**	**Description**	**Page**
$J, J_1, \ldots$	$\in \mathbb{J}_T$	reactive scheduling	elementary intervals for task T	
J_{max}	$\in \mathbb{J}^{\geq\vartheta_u}_{\mathbb{T}',[t_0;t_0+ws[}$	reactive scheduling	interval with latest start time	52
j_T	$\in \mathbb{N}_0$	extended model	maximum release jitter of periodic task T	76
$\mathbb{LT}_{P,I}$	$\equiv \mathbb{N}_0$	extended model	set of local time instants of method instance I with respect to processor P	72
$\mathbb{LT}_{P,I}$	$\equiv \mathbb{N}_0$	extended model	set of local time instants for task instance I with regard to to processor P	84
$\mathbb{LT}_T$	$\equiv \mathbb{N}_0$	basic model	local time for task T	17
$\mathbb{LTF}_{\mathbb{I}'}$	$\equiv \prod\limits_{P\in\mathbb{P},I\in\mathbb{I}'} \mathbb{LTF}_{P,I}$	extended model	set of vectors of local time functions for method instances in $\mathbb{I}'$	73
$\mathbb{LTF}_{\mathbb{I}'}$	$\equiv \prod\limits_{P\in\mathbb{P},I\in\mathbb{I}'} \mathbb{LTF}_{P,I}$	extended model	set of all possible vectors of local time functions for instances in $\mathbb{I}'$	84
$\mathbb{LTF}_{\mathbb{T}'}$	$\equiv \prod\limits_{T\in\mathbb{T}'} \mathbb{LTF}_T$	basic model	set of vectors of local time functions for tasks in $\mathbb{T}'$	21
$\mathbb{LTF}_{P,I}$	$\equiv (\mathbb{LT}_{P,I})^{\mathbb{GT}}$	extended model	set of all possible local time functions for method instance I on processor P	73
$\mathbb{LTF}_{P,I}$	$\subseteq (\mathbb{LT}_{P,I})^{\mathbb{GT}}$	extended model	set of all possible local time functions for I on processor P	84
$\mathbb{LTF}_T$	$\equiv (\mathbb{LT}_T)^{\mathbb{GT}}$	basic model	set of possible local time functions for task T	20
$lLength$	$\in \mathbb{N}_0$	tabu search	maximum length of the tabu list	58
$maxDiv$	$\in \mathbb{N}_0$	tabu search	maximum number of diversification steps	58

Symbol	**Type**	**Context**	**Description**	**Page**
$maxImp$	$\in \mathbb{N}_0$	tabu search	maximum number of normal search steps without noticeable improvement before triggering a diversification step	58
lss	$\in (\mathbb{N}_0)^{\mathbb{N}_0 \times \mathbb{N}_0}$	local search	size of the local search space for one interval	58
$\mathbb{M}$	(enumeration)	extended model	set of all methods	72
$M_1, M_2, \ldots$	$\in \mathbb{M}$	extended model	methods	72
$\mathbb{N}$		general	set of positive natural numbers	
$\mathbb{N}_0$		general	set of natural numbers	
$n, n_1, \ldots$	$\in \mathbb{LT}_T$	basic model	local time instants for task T	
$\mathbb{P}$	(enumeration)	extended model	set of processors	71
$P_1, P_2, \ldots$	$\in \mathbb{P}$	extended model	processors	71
$Pr(s, a, s')$	$\in [0; 1]$	MDPs	probability of going from state s to s' when executing action a	64
per_T	$\in \mathbb{N}_0$	extended model	period length of periodic task T	76
$pred$	$\in (2^{\mathbb{T}})^{\mathbb{T}}$	extended model	predecessor function for task dependency graph	100
$pred$	$\in (2^{\mathbb{I}_{\mathbb{T}}})^{\mathbb{I}_{\mathbb{T}}}$	extended model	successor function for instance dependency graph	102
p_T	$\in [0; 1]$	extended model	release probability of aperiodic task T	77
$p_T(t)$	$\in [0; 1]$	MDPs	probability of task T being released at time t	64
$\mathbb{QF}_{\mathbb{M}'}$	$\equiv \prod_{M \in \mathbb{M}', P \in \mathbb{P}} \mathbb{QF}_{M,P}$	extended model	set of vectors of all possible quality functions for methods in $\mathbb{M}'$	73

Symbol	**Type**	**Context**	**Description**	**Page**
$\mathbb{QF}_{\mathbb{T}'}$	$\equiv \prod\limits_{T\in\mathbb{T}'} \mathbb{QF}_T$	basic model	set of all possible vectors of quality functions for tasks in $\mathbb{T}'$	19
$\mathbb{QF}_{M,P}$	$\equiv (\mathbb{R}_0^+)^{\cup_{k\in\mathbb{N}_0}\mathbb{LT}_{P,I_M^k}}$	extended model	set of possible quality functions for method M on processor P	73
$\mathbb{QF}_T$	$\equiv (\mathbb{R}_0^+)^{\mathbb{LT}_T}$	basic model	set of possible quality functions for task T	19
$\vec{q}$	$\in \mathbb{QF}_{\mathbb{T}'}$	basic model	vector of quality functions for tasks in $\mathbb{T}'$	19
$\vec{q}$	$\in \mathbb{QF}_{\mathbb{M}'}$	extended model	vector of quality functions for methods in $\mathbb{M}'$	73
$q_{P,M}$	$\in \mathbb{R}^{\cup_{k\in\mathbb{N}_0}\mathbb{LT}_{P,I_M^k}}$	extended model	quality function of method M on processor P	72
q_T	$\in (\mathbb{R}_0^+)^{\mathbb{LT}_T}$	basic model	quality function of task T	18
$\mathbb{R}$		general	set of real numbers	
$\mathbb{R}_0^+$		general	set of non-negative real numbers	
R	$\in \mathbb{R}^{\hat{\mathbb{S}}}$	MDPs	reward function	65
r_I	$\in \mathbb{GT}$	extended model	release time for task and method instances	75
r_T	$\in \mathbb{GT}$	basic model	release time of task T	17
$\mathbb{S}$	$\equiv \bigcup\limits_{t\in\mathbb{GT}} \mathbb{S}_t$	MDPs	set of possible states	64
$\mathbb{S}_t$	$\subseteq$ $\mathbb{GT} \times \mathbb{T}(1)\|\ldots\|\mathbb{T}(t) \times 2^{\mathbb{T}}$	MDPs	set of possible states at time t	63
$\hat{\mathbb{S}}$	$\subseteq \mathbb{S}$	MDPs	state envelope	64
$s = (t, \langle a_0, \ldots, a_t\rangle, \mathbb{T}(t))$	$\in \mathbb{S}_t$	MDPs	state at time t	63
sa_i	$\in [0; 1]$	meta scheduling	scheduling allowance for the i-th scheduling phase	122

Symbol	**Type**	**Context**	**Description**	**Page**
s_{err}	$\in \mathbb{S}$	MDPs	state used for illegal transitions	64
$slope_i$	$\in \mathbb{R}_0^+$	meta scheduling	value density: value gain in i-th scheduling phase divided by the length of the partial schedule	123
s_{out}	$\in \mathbb{S}$	MDPs	state used for legal transitions outside the envelope	64
spd	$\in \mathbb{N}$	meta scheduling	window size for derivative component of PID controller	126
spi	$\in \mathbb{N}$	meta scheduling	window size for integral component of PID controller	126
$sProb_J$	$\in [0;1]$	local search	probability of selection of interval J during search	60
ss	$\in (\mathbb{N}_0)^{2^{\mathbb{T}} \times 2^{\mathbb{J}^{\geq \vartheta u}_{\mathbb{T},[t_0;t_0+ws[}}}$	local search	size of the search space within scheduling window	59
$stcd_T$	$\in \mathbb{R}_0^+$	results	weighted average standard deviation of release times of children of task T	171
std_T	$\in \mathbb{R}_0^+$	results	standard deviation of release time for task T	171
std_T^*	$\in \mathbb{R}_0^+$	results	aggregated standard deviation of release time for task T and its children	171
$\mathbb{T}$	(enumeration)	basic model	set of all tasks	15
$\mathbb{T}', \mathbb{T}'', \ldots$	$\subseteq \mathbb{T}$	basic model	task sets	17
$\mathbb{T}_\infty$	$\subseteq \mathbb{T}$	extended model	set of tasks with infinitely many instances	82
$\mathbb{T}_1$	$\subseteq \mathbb{T}$	extended model	set of tasks with exactly one instance	82
$\mathbb{T}'_J$	$\subseteq \mathbb{T}$	reactive scheduling	set of ready tasks within interval J	47

Symbol	**Type**	**Context**	**Description**	**Page**
$\mathbb{T}(t)$	$\subseteq \mathbb{T}$	MDPs	set of tasks released up to time $t \in \mathbb{GT}$	63
$T, T_1, \ldots$	$\in \mathbb{T}$	basic model	tasks	15
$Temp_{end}$	$\in \mathbb{R}_0^+$	simulated annealing	minimum temperature	52
$Temp_{start}$	$\in \mathbb{R}_0^+$	simulated annealing	start temperature	52
$t, t_1, \ldots$	$\in \mathbb{GT}$	basic model	global time instants	
$\mathbb{UF}_{\mathbb{I}'_{\mathbb{T}}}$	$\equiv \prod\limits_{I \in \mathbb{I}'_{\mathbb{T}}} \mathbb{UF}_I$	extended model	set of vectors of all possible utility functions for task instances in $\mathbb{I}'_{\mathbb{T}}$	78
$\mathbb{UF}_{\mathbb{T}'}$	$\equiv \prod\limits_{T \in \mathbb{T}'} \mathbb{UF}_T$	basic model	set of all possible vectors of utility functions for tasks in $\mathbb{T}'$	20
$\mathbb{UF}_I$	$\equiv (\mathbb{R}_0^+)^{\mathbb{GT}}$	extended model	set of possible utility functions for task instance I	78
$\mathbb{UF}_T$	$\equiv (\mathbb{R}_0^+)^{\mathbb{GT}}$	basic model	set of possible utility functions for task T	20
$U_T^{\vartheta_q}$	$\in \mathbb{R}_0^+$	results	threshold utilisation	170
$\vec{u}$	$\in \mathbb{UF}_{\mathbb{T}'}$	basic model	vector of utility functions for tasks in $\mathbb{T}'$	21
u_I	$\in (\mathbb{R}_0^+)^{\mathbb{GT}}$	extended model	utility function for task instance I	78
u_T	$\in (\mathbb{R}_0^+)^{\mathbb{GT}}$	basic model	utility function of task T	19
u_T	$\in (\mathbb{R}_0^+)^{\mathbb{J}^{\geq\vartheta}_{T,[t_0;t_0+ws[,\mathbb{T}'}}$	reactive scheduling	utility function in terms of elementary intervals	48
V_π	$\in \mathbb{R}^{\hat{\mathbb{S}}}$	MDPs	discounted sum of rewards	65
$v_{\vec{q},\vec{u}}$	$\in (\mathbb{R}_0^+)^{\mathbb{LTF}_{\mathbb{T}'} \times \mathbb{GT}}$	basic model	value function for vectors of quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$	22

Symbol	**Type**	**Context**	**Description**	**Page**
$v_{\vec{q},\vec{u}}$	$\in \left(\mathbb{R}_0^+\right)^{\mathbb{LTF}_{\mathbb{T}'} \times \mathbb{J}^{\geq\vartheta}_{T,[t_0;t_0+ws[,\mathbb{T}'}}$	reactive scheduling	value function for vectors of quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ in terms of elementary intervals	48
$v_{\vec{q},\vec{u}}$	$\in \left(\mathbb{R}_0^+\right)^{\mathbb{LTF}_{\mathbb{I}_{\mathbb{M}'}} \times \mathbb{GT}}$	extended model	value function for vectors of quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{I}_{\mathbb{M}'}}$, $\vec{u} \in \mathbb{UF}_{\mathbb{I}_{\mathbb{T}'}}$	79
$v_{I,\vec{q},\vec{u}}$	$\in \left(\mathbb{R}_0^+\right)^{\mathbb{LTF}_{\mathbb{I}'_{\mathbb{T}} \cup \mathbb{I}'_{\mathbb{M}}} \times \mathbb{GT}}$	extended model	value function for task instance $I \in \mathbb{I}'_{\mathbb{T}}$ and vectors of quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{I}_{\mathbb{M}'}}$, $\vec{u} \in \mathbb{UF}_{\mathbb{I}_{\mathbb{T}'}}$	85
$v^*_{I,\vec{q},\vec{u}}$	$\in \left(\mathbb{R}_0^+\right)^{\mathbb{LTF}_{\mathbb{I}'_{\mathbb{T}} \cup \mathbb{I}'_{\mathbb{M}}} \times \mathbb{GT}}$	extended model	value function for task instance $I \in \mathbb{I}'_{\mathbb{T}}$, vectors of quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{I}_{\mathbb{M}'}}$, $\vec{u} \in \mathbb{UF}_{\mathbb{I}_{\mathbb{T}'}}$ and unit value edge weights in dependency graph	104
$v^\ddagger_{I,\vec{q},\vec{u}}$	$\in \left(\mathbb{R}_0^+\right)^{\mathbb{LTF}_{\mathbb{I}'_{\mathbb{T}} \cup \mathbb{I}'_{\mathbb{M}}} \times \mathbb{GT}}$	extended model	value function for task instance $I \in \mathbb{I}'_{\mathbb{T}}$, vectors of quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{I}_{\mathbb{M}'}}$, $\vec{u} \in \mathbb{UF}_{\mathbb{I}_{\mathbb{T}'}}$ and non-unit value edge weights in dependency graph	107
$v^\bullet_{I,\vec{q},\vec{u}}$	$\in \left(\mathbb{R}_0^+\right)^{\mathbb{LTF}_{\mathbb{I}'_{\mathbb{T}} \cup \mathbb{I}'_{\mathbb{M}}} \times \mathbb{GT}}$	extended model	value function for task instance $I \in \mathbb{I}'_{\mathbb{T}}$, vectors of quality and utility functions $\vec{q} \in \mathbb{QF}_{\mathbb{I}_{\mathbb{M}'}}$, $\vec{u} \in \mathbb{UF}_{\mathbb{I}_{\mathbb{T}'}}$, non-unit value edge weights in dependency graph and one-time value flow	107
$weight$	$\in [0;1]^{\mathbb{T} \times \mathbb{T}}$	extended model	weight on dependency edges	101

Appendix B

Proofs

B.1 Sum of Product with Outer Hold Operator

We have to prove that the function

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t'))$$

is consistent with the properties of value functions.

1. global time monotony

   For any $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, \vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$ and $t \in \mathbb{GT}$

$$\begin{array}{rcl} v_{\vec{q},\vec{u}}(\vec{\tau}, t+1) & = & \max_{t' \leq t+1} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\ & = & \max\left(\sum_{T \in \mathbb{T}'} u_T(t+1) \cdot q_T(\tau_T(t+1)), \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t'))\right) \\ & \geq & \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\ & = & v_{\vec{q},\vec{u}}(\vec{\tau}, t) \end{array}$$

2. allocation history monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, t_1 \in \mathbb{GT}, \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   such that

$$\forall t' \in \{0, \ldots, t_1\} : \forall T \in \mathbb{T}' : \tau_T(t') = \tau'_T(t')$$

Then the following is true for $t \in \mathbb{GT}, t \leq t_1$:

$$\begin{aligned}
v_{\vec{q},\vec{u}}(\vec{\tau}, t) &= \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot (q_T(\tau_T(t')) \\
&= \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')) \\
&= v_{\vec{q},\vec{u}}(\vec{\tau'}, t)
\end{aligned}$$

3. allocation amount monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, T' \in \mathbb{T}', \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   with

$$\forall t \in \mathbb{GT} : \forall T \in \mathbb{T}' \backslash \{T'\} : \tau_T(t) = \tau'_T(t)$$

$$\forall t \in \mathbb{GT} : \tau_{T'}(t) \leq \tau'_{T'}(t)$$

   As quality functions are monotonically increasing with local time, and quality and utility values are non-negative, we know that

$$\begin{aligned}
v_{\vec{q},\vec{u}}(\vec{\tau}, t) &= \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\
&= \max_{t' \leq t} \left(u_{T'}(t') \cdot q_{T'}(\tau_{T'}(t)) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} u_T(t') \cdot q_T(\tau_T(t')) \right) \\
&= \max_{t' \leq t} \left(u_{T'}(t') \cdot q_{T'}(\tau_{T'}(t)) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} u_T(t') \cdot q_T(\tau'_T(t')) \right) \\
&\leq \max_{t' \leq t} \left(u_{T'}(t') \cdot q_{T'}(\tau'_{T'}(t)) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} u_T(t') \cdot q_T(\tau'_T(t')) \right) \\
&= \max_{t' \leq t} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')) \\
&= v_{\vec{q},\vec{u}}(\vec{\tau'}, t)
\end{aligned}$$

4. allocation time monotony

   follows from 3

5. reducibility to utility intervals

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ and $\vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$ with

$$\forall T \in \mathbb{T}' : u_T(t_1) = \dots u_T(t_2)$$

$$\forall t \leq t_1 : \forall T \in \mathbb{T}' : \tau'_T(t) = \tau_T(t)$$

and

$$\forall T \in \mathbb{T}' : \tau'_T(t_2) = \tau_T(t_2)$$

Then

$$\begin{aligned}
v_{\vec{q},\vec{u}}(\vec{\tau}, t_2) &= \max_{t' \leq t_2} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\
&= \max\Big(\max_{t' < t_1} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')), \max_{t_1 \leq t' \leq t_2} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \Big) \\
&= \max\Big(\max_{t' < t_1} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{t_1 \leq t' \leq t_2} \sum_{T \in \mathbb{T}'} u_T(t_2) \cdot q_T(\tau_T(t')) \Big) \\
&= \max\Big(\max_{t' < t_1} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), u_T(t_2) \cdot \sum_{T \in \mathbb{T}'} q_T(\tau_T(t_2)) \Big) \\
&= \max\Big(\max_{t' < t_1} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), u_T(t_2) \cdot \sum_{T \in \mathbb{T}'} q_T(\tau'_T(t_2)) \Big) \\
&= \max\Big(\max_{t' < t_1} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{t_1 \leq t' \leq t_2} \sum_{T \in \mathbb{T}'} u_T(t_2) \cdot q_T(\tau'_T(t')) \Big) \\
&= \max\Big(\max_{t' < t_1} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{t_1 \leq t' \leq t_2} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')) \Big) \\
&= \max_{t' \leq t_2} \sum_{T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')) \\
&= v_{\vec{q},\vec{u}}(\vec{\tau'}, t_2)
\end{aligned}$$

6. utility monotony

   Let $\mathbb{T}', \mathbb{T}'' \subseteq \mathbb{T}$ with $T' \in \mathbb{T}', T'' \in \mathbb{T}''$ and $\mathbb{T}' \backslash \{T'\} = \mathbb{T}'' \backslash \{T''\}$

   and

   $\vec{q'} \in \mathbb{QF}_{\mathbb{T}'}, \vec{q''} \in \mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}, \vec{\tau''} \in \mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'} \in \mathbb{UF}_{\mathbb{T}'}, \vec{u''} \in \mathbb{UF}_{\mathbb{T}''}$, with

$$r'_{T'} = r''_{T''}, q'_{T'} = q''_{T''}, \tau'_{T'} = \tau''_{T''}$$

$$\forall t \in \mathbb{GT} : u'_{T'}(t) \leq u''_{T''}(t)$$

As qualities are non-negative, we know that

$$\begin{aligned}
v_{\vec{q'},\vec{u'}}(\vec{\tau'},t) &= \max_{t'\le t}\sum_{T\in\mathbb{T}'} u'_T(t')\cdot q'_T(\tau'_T(t')) \\
&= \max_{t'\le t}\left(u'_{T'}(t')\cdot q'_{T'}(\tau'_{T'}(t')) + \sum_{T\in\mathbb{T}'\backslash\{T'\}} u'_T(t')\cdot q'_T(\tau'_T(t'))\right) \\
&= \max_{t'\le t}\left(u'_{T'}(t')\cdot q''_{T''}(\tau''_{T''}(t')) + \sum_{T\in\mathbb{T}''\backslash\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\right) \\
&\le \max_{t'\le t}\left(u''_{T''}(t')\cdot q''_{T''}(\tau''_{T''}(t')) + \sum_{T\in\mathbb{T}''\backslash\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\right) \\
&= \max_{t'\le t}\sum_{T\in\mathbb{T}''} u''_T(t')\cdot q''_T(\tau''_T(t')) \\
&= v_{\vec{q''},\vec{u''}}(\vec{\tau''},t)
\end{aligned}$$

7. quality monotony

   Let $\mathbb{T}',\mathbb{T}''\subseteq\mathbb{T}$ with $T'\in\mathbb{T}',T''\in\mathbb{T}''$ and $\mathbb{T}'\backslash\{T'\}=\mathbb{T}''\backslash\{T''\}$

   and

   $\vec{q'}\in\mathbb{QF}_{\mathbb{T}'},\vec{q''}\in\mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'}\in\mathbb{LTF}_{\mathbb{T}'},\vec{\tau''}\in\mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'}\in\mathbb{UF}_{\mathbb{T}'},\vec{u''}\in\mathbb{UF}_{\mathbb{T}''}$, with

$$r'_{T'}=r''_{T''},u'_{T'}=u''_{T''},\tau'_{T'}=\tau''_{T''}$$

$$\forall n'\in\mathbb{LT}_{T'},n''\in\mathbb{LT}_{T''}:n'\equiv n''\Rightarrow q'_{T'}(n')\le q''_{T''}(n'')$$

   As utilities are non-negative, we know that

$$\begin{aligned}
v_{\vec{q'},\vec{u'}}(\vec{\tau'},t) &= \max_{t'\le t}\sum_{T\in\mathbb{T}'} u'_T(t')\cdot q'_T(\tau'_T(t')) \\
&= \max_{t'\le t}\left(u'_{T'}(t')\cdot q'_{T'}(\tau'_{T'}(t')) + \sum_{T\in\mathbb{T}'\backslash\{T'\}} u'_T(t')\cdot q'_T(\tau'_T(t'))\right) \\
&= \max_{t'\le t}\left(u''_{T''}(t')\cdot q'_{T'}(\tau''_{T''}(t')) + \sum_{T\in\mathbb{T}''\backslash\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\right) \\
&\le \max_{t'\le t}\left(u''_{T''}(t')\cdot q''_{T''}(\tau''_{T''}(t')) + \sum_{T\in\mathbb{T}''\backslash\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\right)
\end{aligned}$$

$$\begin{aligned} &= \max_{t' \leq t} \sum_{T \in \mathbb{T}''} u''_T(t') \cdot q''_T(\tau''_T(t')) \\ &= v_{\vec{q''}, \vec{u''}}(\vec{\tau''}, t) \end{aligned}$$

□

B.2 Sum of Product with Inner Hold Operator

We have to prove that the function

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \sum_{T \in \mathbb{T}'} \max_{t' \leq t} u_T(t') \cdot q_T(\tau_T(t'))$$

is consistent with the properties of value functions.

1. global time monotony

   For any $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, \vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$ and $t \in \mathbb{GT}$

$$\begin{aligned} v_{\vec{q},\vec{u}}(\vec{\tau}, t+1) &= \sum_{T \in \mathbb{T}'} \max_{t' \leq t+1} u_T(t') \cdot q_T(\tau_T(t')) \\ &= \sum_{T \in \mathbb{T}'} \max \left(\max_{t' \leq t} u_T(t') \cdot q_T(\tau_T(t')), u_T(t+1) \cdot q_T(\tau_T(t+1)) \right) \\ &\geq \sum_{T \in \mathbb{T}'} \max_{t' \leq t} u_T(t') \cdot q_T(\tau_T(t')) \\ &= v_{\vec{q},\vec{u}}(\vec{\tau}, t) \end{aligned}$$

2. allocation history monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, t_1 \in \mathbb{GT}, \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   such that

$$\forall t' \in \{0, \ldots, t_1\} : \forall T \in \mathbb{T}' : \tau_T(t') = \tau'_T(t')$$

   Then the following is true for $t \in \mathbb{GT}, t \leq t_1$:

$$\begin{aligned} v_{\vec{q},\vec{u}}(\vec{\tau}, t) &= \sum_{T \in \mathbb{T}'} \max_{t' \leq t} u_T(t') \cdot q_T(\tau_T(t')) \\ &= \sum_{T \in \mathbb{T}'} \max_{t' \leq t} u_T(t') \cdot q_T(\tau'_T(t')) \\ &= v_{\vec{q},\vec{u}}(\vec{\tau'}, t) \end{aligned}$$

3. allocation amount monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, T' \in \mathbb{T}', \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   with

   $$\forall t \in \mathbb{GT} : \forall T \in \mathbb{T}' \backslash \{T'\} : \tau_T(t) = \tau'_T(t)$$

   $$\forall t \in \mathbb{GT} : \tau_{T'}(t) \leq \tau'_{T'}(t)$$

   As quality functions are monotonically increasing with local time, and quality and utility values are non-negative, we know that

   $$\begin{aligned}
   v_{\vec{q},\vec{u}}(\vec{\tau}, t) &= \sum_{T \in \mathbb{T}'} \max_{t' \leq t} u_T(t') \cdot q_T(\tau_T(t')) \\
   &= \max_{t' \leq t} (u_{T'}(t') \cdot q_{T'}(\tau_{T'}(t))) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} \max_{t' \leq t} (u_T(t') \cdot q_T(\tau_T(t'))) \\
   &= \max_{t' \leq t} (u_{T'}(t') \cdot q_{T'}(\tau_{T'}(t))) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} \max_{t' \leq t} (u_T(t') \cdot q_T(\tau'_T(t'))) \\
   &\leq \max_{t' \leq t} (u_{T'}(t') \cdot q_{T'}(\tau'_{T'}(t))) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} \max_{t' \leq t} (u_T(t') \cdot q_T(\tau'_T(t'))) \\
   &= \sum_{T \in \mathbb{T}'} \max_{t' \leq t} u_T(t') \cdot q_T(\tau'_T(t')) \\
   &= v_{\vec{q},\vec{u}}(\vec{\tau'}, t)
   \end{aligned}$$

4. allocation time monotony

   follows from 3

5. reducibility to utility intervals

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ and $\vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$ with

   $$\forall T \in \mathbb{T}' : u_T(t_1) = \ldots u_T(t_2)$$

   $$\forall t \leq t_1 : \forall T \in \mathbb{T}' : \tau'_T(t) = \tau_T(t)$$

   and

   $$\forall T \in \mathbb{T}' : \tau'_T(t_2) = \tau_T(t_2)$$

Then

$$\begin{aligned}
v_{\vec{q},\vec{u}}(\vec{\tau},t_2) &= \sum_{T\in\mathbb{T}'} \max_{t'\leq t_2} u_T(t')\cdot q_T(\tau_T(t')) \\
&= \sum_{T\in\mathbb{T}'} \max\Big(\max_{t'<t_1} u_T(t')\cdot q_T(\tau_T(t')), \max_{t_1\leq t'\leq t_2} u_T(t')\cdot q_T(\tau_T(t'))\Big) \\
&= \sum_{T\in\mathbb{T}'} \max\Big(\max_{t'<t_1} u_T(t')\cdot q_T(\tau'_T(t')), \max_{t_1\leq t'\leq t_2} u_T(t_2)\cdot q_T(\tau_T(t'))\Big) \\
&= \sum_{T\in\mathbb{T}'} \max\Big(\max_{t'<t_1} u_T(t')\cdot q_T(\tau'_T(t')), u_T(t_2)\cdot q_T(\tau_T(t_2))\Big) \\
&= \sum_{T\in\mathbb{T}'} \max\Big(\max_{t'<t_1} u_T(t')\cdot q_T(\tau'_T(t')), u_T(t_2)\cdot q_T(\tau'_T(t_2))\Big) \\
&= \sum_{T\in\mathbb{T}'} \max\Big(\max_{t'<t_1} u_T(t')\cdot q_T(\tau'_T(t')), \max_{t_1\leq t'\leq t_2} u_T(t_2)\cdot q_T(\tau'_T(t'))\Big) \\
&= \sum_{T\in\mathbb{T}'} \max\Big(\max_{t'<t_1} u_T(t')\cdot q_T(\tau'_T(t')), \max_{t_1\leq t'\leq t_2} u_T(t')\cdot q_T(\tau'_T(t'))\Big) \\
&= \sum_{T\in\mathbb{T}'} \max_{t'\leq t_2} u_T(t')\cdot q_T(\tau'_T(t')) \\
&= v_{\vec{q},\vec{u}}(\vec{\tau'},t_2)
\end{aligned}$$

6. utility monotony

   Let $\mathbb{T}',\mathbb{T}''\subseteq\mathbb{T}$ with $T'\in\mathbb{T}',T''\in\mathbb{T}''$ and $\mathbb{T}'\backslash\{T'\}=\mathbb{T}''\backslash\{T''\}$

   and

   $\vec{q'}\in\mathbb{QF}_{\mathbb{T}'},\vec{q''}\in\mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'}\in\mathbb{LTF}_{\mathbb{T}'},\vec{\tau''}\in\mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'}\in\mathbb{UF}_{\mathbb{T}'},\vec{u''}\in\mathbb{UF}_{\mathbb{T}''}$, with

$$r'_{T'}=r''_{T''}, q'_{T'}=q''_{T''}, \tau'_{T'}=\tau''_{T''}$$

$$\forall t\in\mathbb{GT}: u'_{T'}(t)\leq u''_{T''}(t)$$

   As qualities are non-negative, we know that

$$\begin{aligned}
v_{\vec{q'},\vec{u'}}(\vec{\tau'},t) &= \sum_{T\in\mathbb{T}'} \max_{t'\leq t} u'_T(t')\cdot q'_T(\tau'_T(t')) \\
&= \max_{t'\leq t}\left(u'_{T'}(t')\cdot q'_{T'}(\tau'_{T'}(t'))\right) + \sum_{T\in\mathbb{T}'\backslash\{T'\}} \max_{t'\leq t}\left(u'_T(t')\cdot q'_T(\tau'_T(t'))\right) \\
&= \max_{t'\leq t}\left(u'_{T'}(t')\cdot q''_{T''}(\tau''_{T''}(t'))\right) + \sum_{T\in\mathbb{T}''\backslash\{T''\}} \max_{t'\leq t}\left(u''_T(t')\cdot q''_T(\tau''_T(t'))\right)
\end{aligned}$$

$$
\begin{aligned}
&\leq \max_{t'\leq t}\left(u''_{T''}(t')\cdot q''_{T''}(\tau''_{T''}(t'))\right) + \sum_{T\in\mathbb{T}''\setminus\{T''\}} \max_{t'\leq t}\left(u''_T(t')\cdot q''_T(\tau''_T(t'))\right)\\
&= \sum_{T\in\mathbb{T}''}\max_{t'\leq t} u''_T(t')\cdot q''_T(\tau''_T(t'))\\
&= v_{\vec{q''},\vec{u''}}(\vec{\tau''},t)
\end{aligned}
$$

7. quality monotony

   Let $\mathbb{T}',\mathbb{T}''\subseteq\mathbb{T}$ with $T'\in\mathbb{T}',T''\in\mathbb{T}''$ and $\mathbb{T}'\setminus\{T'\}=\mathbb{T}''\setminus\{T''\}$

   and

   $\vec{q'}\in\mathbb{QF}_{\mathbb{T}'},\vec{q''}\in\mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'}\in\mathbb{LTF}_{\mathbb{T}'},\vec{\tau''}\in\mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'}\in\mathbb{UF}_{\mathbb{T}'},\vec{u''}\in\mathbb{UF}_{\mathbb{T}''}$, with

   $$r'_{T'}=r''_{T''},u'_{T'}=u''_{T''},\tau'_{T'}=\tau''_{T''}$$

   $$\forall n'\in\mathbb{LT}_{T'},n''\in\mathbb{LT}_{T''}: n'\equiv n''\Rightarrow q'_{T'}(n')\leq q''_{T''}(n'')$$

   As utilities are non-negative, we know that

   $$
   \begin{aligned}
   v_{\vec{q'},\vec{u'}}(\vec{\tau'},t) &= \sum_{T\in\mathbb{T}'}\max_{t'\leq t} u'_T(t')\cdot q'_T(\tau'_T(t'))\\
   &= \max_{t'\leq t}\left(u'_{T'}(t')\cdot q'_{T'}(\tau'_{T'}(t'))\right) + \sum_{T\in\mathbb{T}'\setminus\{T'\}}\max_{t'\leq t}\left(u'_T(t')\cdot q'_T(\tau'_T(t'))\right)\\
   &= \max_{t'\leq t}\left(u''_{T''}(t')\cdot q'_{T'}(\tau'_{T'}(t'))\right) + \sum_{T\in\mathbb{T}''\setminus\{T''\}}\max_{t'\leq t}\left(u''_T(t')\cdot q''_T(\tau''_T(t'))\right)\\
   &\leq \max_{t'\leq t}\left(u''_{T''}(t')\cdot q''_{T''}(\tau''_{T''}(t'))\right) + \sum_{T\in\mathbb{T}''\setminus\{T''\}}\max_{t'\leq t}\left(u''_T(t')\cdot q''_T(\tau''_T(t'))\right)\\
   &= \sum_{T\in\mathbb{T}''}\max_{t'\leq t} u''_T(t')\cdot q''_T(\tau''_T(t'))\\
   &= v_{\vec{q''},\vec{u''}}(\vec{\tau''},t)
   \end{aligned}
   $$

□

B.3 Sum of Product with Additional Conditions

We have to prove that the function

$$v_{\vec{q},\vec{u}}(\vec{\tau},t)=\sum_{T\in\mathbb{T}'}u_T(t)\cdot q_T(\tau_T(t))$$

with the constraint

$$\forall T \in \mathbb{T} : \forall t \in \mathbb{GT} : u_T(t+1) \cdot q_T(\tau_T(t+1)) \geq u_T(t) \cdot q_T(\tau_T(t))$$

is consistent with the properties of value functions.

1. global time monotony

   For any $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, \vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$ and $t \in \mathbb{GT}$

$$\begin{aligned} v_{\vec{q},\vec{u}}(\vec{\tau}, t+1) &= \sum_{T \in \mathbb{T}'} u_T(t+1) \cdot q_T(\tau_T(t+1)) \\ &\geq \sum_{T \in \mathbb{T}'} u_T(t) \cdot q_T(\tau_T(t)) \\ &= v_{\vec{q},\vec{u}}(\vec{\tau}, t) \end{aligned}$$

2. allocation history monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, t_1 \in \mathbb{GT}, \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   such that

$$\forall t' \in \{0, \ldots, t_1\} : \forall T \in \mathbb{T}' : \tau_T(t') = \tau'_T(t')$$

   Then the following is true for $t \in \mathbb{GT}, t \leq t_1$:

$$\begin{aligned} v_{\vec{q},\vec{u}}(\vec{\tau}, t) &= \sum_{T \in \mathbb{T}'} u_T(t) \cdot q_T(\tau_T(t)) \\ &= \sum_{T \in \mathbb{T}'} u_T(t) \cdot q_T(\tau'_T(t)) \\ &= v_{\vec{q},\vec{u}}(\vec{\tau'}, t) \end{aligned}$$

3. allocation amount monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, T' \in \mathbb{T}', \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   with

$$\forall t \in \mathbb{GT} : \forall T \in \mathbb{T}' \backslash \{T'\} : \tau_T(t) = \tau'_T(t)$$
$$\forall t \in \mathbb{GT} : \tau_{T'}(t) \leq \tau'_{T'}(t)$$

   As quality functions are monotonically increasing with local time, and quality and utility values are non-negative, we know that

$$\begin{aligned}
v_{\vec{q},\vec{u}}(\vec{\tau},t) &= \sum_{T\in\mathbb{T}'} u_T(t)\cdot q_T(\tau_T(t)) \\
&= u_{T'}(t)\cdot q_{T'}(\tau_{T'}(t)) + \sum_{T\in\mathbb{T}'\setminus\{T'\}} u_T(t)\cdot q_T(\tau_T(t)) \\
&= u_{T'}(t)\cdot q_{T'}(\tau_{T'}(t)) + \sum_{T\in\mathbb{T}'\setminus\{T'\}} u_T(t)\cdot q_T(\tau'_T(t)) \\
&\leq u_{T'}(t)\cdot q_{T'}(\tau'_{T'}(t)) + \sum_{T\in\mathbb{T}'\setminus\{T'\}} u_T(t)\cdot q_T(\tau'_T(t)) \\
&= \sum_{T\in\mathbb{T}'} u_T(t)\cdot q_T(\tau'_T(t)) \\
&= v_{\vec{q},\vec{u}}(\vec{\tau'},t)
\end{aligned}$$

4. allocation time monotony

   follows from 3

5. reducibility to utility intervals

   Let $\vec{q}\in\mathbb{QF}_{\mathbb{T}'}$, $\vec{u}\in\mathbb{UF}_{\mathbb{T}'}$ and $\vec{\tau},\vec{\tau'}\in\mathbb{LTF}_{\mathbb{T}'}$ with

$$\forall T\in\mathbb{T}' : u_T(t_1) = \ldots u_T(t_2)$$

$$\forall t\leq t_1 : \forall T\in\mathbb{T}' : \tau'_T(t) = \tau_T(t)$$

   and

$$\forall T\in\mathbb{T}' : \tau'_T(t_2) = \tau_T(t_2)$$

   Then

$$\begin{aligned}
v_{\vec{q},\vec{u}}(\vec{\tau},t_2) &= \sum_{T\in\mathbb{T}'} u_T(t_2)\cdot q_T(\tau_T(t_2)) \\
&= \sum_{T\in\mathbb{T}'} u_T(t_2)\cdot q_T(\tau'_T(t_2)) \\
&= v_{\vec{q},\vec{u}}(\vec{\tau'},t_2)
\end{aligned}$$

6. utility monotony

   Let $\mathbb{T}', \mathbb{T}'' \subseteq \mathbb{T}$ with $T' \in \mathbb{T}', T'' \in \mathbb{T}''$ and $\mathbb{T}' \backslash \{T'\} = \mathbb{T}'' \backslash \{T''\}$

   and

   $\vec{q'} \in \mathbb{QF}_{\mathbb{T}'}, \vec{q''} \in \mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}, \vec{\tau''} \in \mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'} \in \mathbb{UF}_{\mathbb{T}'}, \vec{u''} \in \mathbb{UF}_{\mathbb{T}''}$, with

$$r'_{T'} = r''_{T''}, q'_{T'} = q''_{T''}, \tau'_{T'} = \tau''_{T''}$$

$$\forall t \in \mathbb{GT} : u'_{T'}(t) \leq u''_{T''}(t)$$

   As qualities are non-negative, we know that

$$\begin{aligned}
v_{\vec{q'},\vec{u'}}(\vec{\tau'}, t) &= \sum_{T \in \mathbb{T}'} u'_T(t) \cdot q'_T(\tau'_T(t)) \\
&= u'_{T'}(t) \cdot q'_{T'}(\tau'_{T'}(t)) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} u'_T(t) \cdot q'_T(\tau'_T(t)) \\
&= u'_{T'}(t) \cdot q''_{T''}(\tau''_{T''}(t)) + \sum_{T \in \mathbb{T}'' \backslash \{T''\}} u''_T(t) \cdot q''_T(\tau''_T(t)) \\
&\leq u''_{T''}(t) \cdot q''_{T''}(\tau''_{T''}(t)) + \sum_{T \in \mathbb{T}'' \backslash \{T''\}} u''_T(t) \cdot q''_T(\tau''_T(t)) \\
&= \sum_{T \in \mathbb{T}''} u''_T(t') \cdot q''_T(\tau''_T(t) \\
&= v_{\vec{q''},\vec{u''}}(\vec{\tau''}, t)
\end{aligned}$$

7. quality monotony

   Let $\mathbb{T}', \mathbb{T}'' \subseteq \mathbb{T}$ with $T' \in \mathbb{T}', T'' \in \mathbb{T}''$ and $\mathbb{T}' \backslash \{T'\} = \mathbb{T}'' \backslash \{T''\}$

   and

   $\vec{q'} \in \mathbb{QF}_{\mathbb{T}'}, \vec{q''} \in \mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}, \vec{\tau''} \in \mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'} \in \mathbb{UF}_{\mathbb{T}'}, \vec{u''} \in \mathbb{UF}_{\mathbb{T}''}$, with

$$r'_{T'} = r''_{T''}, u'_{T'} = u''_{T''}, \tau'_{T'} = \tau''_{T''}$$

$$\forall n' \in \mathbb{LT}_{T'}, n'' \in \mathbb{LT}_{T''} : n' \equiv n'' \Rightarrow q'_{T'}(n') \leq q''_{T''}(n'')$$

   As utilities are non-negative, we know that

$$\begin{aligned}
v_{\vec{q'},\vec{u'}}(\vec{\tau'}, t) &= \sum_{T \in \mathbb{T}'} u'_T(t) \cdot q'_T(\tau'_T(t)) \\
&= u'_{T'}(t) \cdot q'_{T'}(\tau'_{T'}(t)) + \sum_{T \in \mathbb{T}' \backslash \{T'\}} u'_T(t) \cdot q'_T(\tau'_T(t))
\end{aligned}$$

$$
\begin{aligned}
&= u''_{T''}(t) \cdot q'_{T'}(\tau'_{T'}(t)) + \sum_{T \in \mathbb{T}'' \setminus \{T''\}} u''_T(t) \cdot q''_T(\tau''_T(t)) \\
&\leq u''_{T''}(t) \cdot q''_{T''}(\tau''_{T''}(t)) + \sum_{T \in \mathbb{T}'' \setminus \{T''\}} u''_T(t) \cdot q''_T(\tau''_T(t)) \\
&= \sum_{T \in \mathbb{T}''} u''_T(t') \cdot q''_T(\tau''_T(t) \\
&= v_{\vec{q''},\vec{u''}}(\vec{\tau''}, t)
\end{aligned}
$$

□

B.4 Tasks with Mandatory and Optional Service Times

We want to show that the definitions of the quality and utility functions, together with the service guarantees, fulfill the requirements for the aforementioned kind of value functions.

For the product of quality and utility the following is true:

1. $t \neq 2^i - 1$ for all $i \in \mathbb{N}, i \geq 1$
   $\Rightarrow t < 2^i - 1 \vee t > 2^i - 1$
   $\Rightarrow (t + 1 < 2^i \wedge t < 2^i) \vee (t + 1 \geq 2^i \wedge t \geq 2^i)$
   $\Rightarrow (u_{T_i}(t + 1) = u_{T_i}(t) = 1) \vee (u_{T_i}(t + 1) = u_{T_i}(t) = \frac{1}{2})$
   $\Rightarrow \Big(u_{T_i}(t + 1) \cdot q_{T_i}(\tau_{T_i}(t + 1)) - u_{T_i}(t) \cdot q_{T_i}(\tau_{T_i}(t)) = q_{T_i}(\tau_{T_i}(t + 1)) - q_{T_i}(\tau_{T_i}(t))\Big) \vee$
   $\Big(u_{T_i}(t+1) \cdot q_{T_i}(\tau_{T_i}(t+1)) - u_{T_i}(t - r_{T_i}) \cdot q_{T_i}(\tau_{T_i}(t)) = \frac{1}{2} q_{T_i}(\tau_{T_i}(t+1)) - \frac{1}{2} q_{T_i}(\tau_{T_i}(t))\Big)$

   It remains to show that $q_{T_i}(\tau_{T_i}(t + 1)) - q_{T_i}(\tau_{T_i}(t)) \geq 0$.

   (a) $\tau_{T_i}(t + 1) = \tau_{T_i}(t)$
   $q_{T_i}(\tau_{T_i}(t + 1)) - q_{T_i}(\tau_{T_i}(t)) = q_{T_i}(\tau_{T_i}(t)) - q_{T_i}(\tau_{T_i}(t)) = 0$

   (b) $\tau_{T_i}(t + 1) = \tau_{T_i}(t) + 1$

      i. $\tau_{T_i}(t) = 0 \Rightarrow \tau_{T_i}(t + 1) = 1$
      $\Rightarrow q_{T_i}(\tau_{T_i}(t + 1)) - q_{T_i}(\tau_{T_i}(t)) = q_{T_i}(1) - q_{T_i}(0) = (\frac{1}{3})^{2^i - 1} - 0 > 0$

      ii. $0 < \tau_{T_i}(t) < 2^{i+1} \Rightarrow 0 < \tau_{T_i}(t + 1) \leq 2^{i+1}$
      $\Rightarrow q_{T_i}(\tau_{T_i}(t)) = (\frac{1}{3})^{2^{i - \tau_{T_i}(t)}} \wedge q_{T_i}(\tau_{T_i}(t + 1)) = (\frac{1}{3})^{2^{i - \tau_{T_i}(t+1)}}$
      $\Rightarrow q_{T_i}(\tau_i(t + 1)) = (\frac{1}{3})^{2^i - \tau_{T_i}(t) - 1} = (\frac{1}{3})^{-1} \cdot (\frac{1}{3})^{2^i - \tau_{T_i}(t)}$
      $= 3 \cdot (\frac{1}{3})^{2^i - \tau_{T_i}(t)} = 3 \cdot q_{T_i}(\tau_{T_i}(t))$
      $\Rightarrow q_{T_i}(\tau_{T_i}(t + 1)) - q_{T_i}(\tau_{T_i}(t)) = 2 \cdot q_{T_i}(\tau_{T_i}(t)) \geq 0$

iii. $\tau_{T_i}(t) \geq 2^{i+1} \Rightarrow \tau_{T_i}(t+1) \geq 2^{i+1}$
$\Rightarrow q_{T_i}(\tau_{T_i}(t+1)) = q_{T_i}(\tau_{T_i}(t)) = 1$
$\Rightarrow q_{T_i}(\tau_{T_i}(t+1)) - q_{T_i}(\tau_{T_i}(t)) = 0$

2. $t = 2^i - 1$
$\tau_{T_i}(t+1) - \tau_{T_i}(t) = \tau_{T_i}(2^i) - \tau_{T_i}(2^i - 1) = \alpha_{T_i}(\tau_{T_i}, 2^i - 1) = 1$
$t + 1 \leq 2^i \wedge t \leq 2^i \Rightarrow \tau_{T_i}(t+1) \leq 2^i \wedge \tau_{T_i}(t) \leq 2^i$
$\frac{q_{T_i}(\tau_{T_i}(t+1))}{q_{T_i}(\tau_{T_i}(t))} = \frac{q_{T_i}(\tau_{T_i}(t)+1)}{q_{T_i}(\tau_{T_i}(t))} = \frac{(\frac{1}{3})^{2^i - \tau_{T_i}(t) - 1}}{(\frac{1}{3})^{2^i - \tau_{T_i}(t)}} = \frac{(\frac{1}{3})^{-1}}{(\frac{1}{3})^0} = 3$
$\frac{u_{T_i}(t+1)}{u_{T_i}(t)} = \frac{u_{T_i}(2^i)}{u_{T_i}(2^i - 1)} = \frac{\frac{1}{2}}{1} = \frac{1}{2}$
$\Rightarrow u_{T_i}(t+1) \cdot q_{T_i}(\tau_{T_i}(t+1)) - u_{T_i}(t) \cdot q_{T_i}(\tau_{T_i}(t))$
$= \frac{1}{2} u_{T_i}(t) \cdot 3 q_{T_i}(\tau_{T_i}(t)) - u_{T_i}(t) \cdot q_{T_i}(\tau_{T_i}(t))$
$= \frac{3}{2} u_{T_i}(t) \cdot q_{T_i}(\tau_{T_i}(t)) - u_{T_i}(t) \cdot q_{T_i}(\tau_{T_i}(t))$
$= \frac{1}{2} u_{T_i}(t) \cdot q_{T_i}(\tau_{T_i}(t)) \geq 0$

$\square$

B.5 Maximum of Product with Hold Operator

We have to prove that the function

$$v_{\vec{q},\vec{u}}(\vec{\tau}, t) := \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t'))$$

is consistent with the properties of value functions.

1. global time monotony

   For any $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, \vec{\tau} \in \mathbb{LTF}_{\mathbb{T}'}$ and $t \in \mathbb{GT}$

$$\begin{aligned} v_{\vec{q},\vec{u}}(\vec{\tau}, t+1) &= \max_{t' \leq t+1, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\ &= \max \left(\max_{T \in \mathbb{T}'} u_T(t+1) \cdot q_T(\tau_T(t+1)), \right. \\ &\qquad \left. \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \right) \\ &\geq \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\ &= v_{\vec{q},\vec{u}}(\vec{\tau}, t) \end{aligned}$$

2. allocation history monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, t_1 \in \mathbb{GT}, \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   such that

   $$\forall t' \in \{0, \ldots, t_1\} : \forall T \in \mathbb{T}' : \tau_T(t') = \tau'_T(t')$$

   Then the following is true for $t \in \mathbb{GT}, t \leq t_1$:

   $$\begin{aligned} v_{\vec{q},\vec{u}}(\vec{\tau}, t) &= \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\ &= \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')) \\ &= v_{\vec{q},\vec{u}}(\vec{\tau'}, t) \end{aligned}$$

3. allocation amount monotony

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}, \vec{u} \in \mathbb{UF}_{\mathbb{T}'}, T' \in \mathbb{T}', \vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$

   with

   $$\forall t \in \mathbb{GT} : \forall T \in \mathbb{T}' \backslash \{T'\} : \tau_T(t) = \tau'_T(t)$$
   $$\forall t \in \mathbb{GT} : \tau_{T'}(t) \leq \tau'_{T'}(t)$$

   As quality functions are monotonically increasing with local time, and quality and utility values are non-negative, we know that

   $$\begin{aligned} v_{\vec{q},\vec{u}}(\vec{\tau}, t) &= \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\ &= \max_{t' \leq t} \Big(\max \Big(u_{T'}(t') \cdot q_{T'}(\tau_{T'}(t)), \max_{T \in \mathbb{T}' \backslash \{T'\}} u_T(t') \cdot q_T(\tau_T(t')) \Big) \Big) \\ &= \max_{t' \leq t} \Big(\max \Big(u_{T'}(t') \cdot q_{T'}(\tau_{T'}(t)), \max_{T \in \mathbb{T}' \backslash \{T'\}} u_T(t') \cdot q_T(\tau'_T(t')) \Big) \Big) \\ &\leq \max_{t' \leq t} \Big(\max \Big(u_{T'}(t') \cdot q_{T'}(\tau'_{T'}(t)), \max_{T \in \mathbb{T}' \backslash \{T'\}} u_T(t') \cdot q_T(\tau'_T(t')) \Big) \Big) \\ &= \max_{t' \leq t, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')) \\ &= v_{\vec{q},\vec{u}}(\vec{\tau'}, t) \end{aligned}$$

4. allocation time monotony

   follows from 3

5. reducibility to utility intervals

   Let $\vec{q} \in \mathbb{QF}_{\mathbb{T}'}$, $\vec{u} \in \mathbb{UF}_{\mathbb{T}'}$ and $\vec{\tau}, \vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}$ with

$$\forall T \in \mathbb{T}' : u_T(t_1) = \ldots u_T(t_2)$$

$$\forall t \leq t_1 : \forall T \in \mathbb{T}' : \tau'_T(t) = \tau_T(t)$$

   and

$$\forall T \in \mathbb{T}' : \tau'_T(t_2) = \tau_T(t_2)$$

   Then

$$\begin{aligned}
v_{\vec{q},\vec{u}}(\vec{\tau}, t_2) &= \max_{t' \leq t_2, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')) \\
&= \max\Big(\max_{t' < t_1, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t')), \max_{t_1 \leq t' \leq t_2, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau_T(t'))\Big) \\
&= \max\Big(\max_{t' < t_1, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{T \in \mathbb{T}'}(u_T(t_2) \cdot \max_{t_1 \leq t' \leq t_2} q_T(\tau_T(t')))\Big) \\
&= \max\Big(\max_{t' < t_1, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{T \in \mathbb{T}'}(u_T(t_2) \cdot q_T(\tau_T(t_2)))\Big) \\
&= \max\Big(\max_{t' < t_1, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{T \in \mathbb{T}'}(u_T(t_2) \cdot q_T(\tau'_T(t_2)))\Big) \\
&= \max\Big(\max_{t' < t_1, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{T \in \mathbb{T}'}(u_T(t_2) \cdot \max_{t_1 \leq t' \leq t_2} q_T(\tau'_T(t')))\Big) \\
&= \max\Big(\max_{t' < t_1, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')), \max_{t_1 \leq t' \leq t_2, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t'))\Big) \\
&= \max_{t' \leq t_2, T \in \mathbb{T}'} u_T(t') \cdot q_T(\tau'_T(t')) \\
&= v_{\vec{q},\vec{u}}(\vec{\tau'}, t_2)
\end{aligned}$$

6. utility monotony

   Let $\mathbb{T}', \mathbb{T}'' \subseteq \mathbb{T}$ with $T' \in \mathbb{T}', T'' \in \mathbb{T}''$ and $\mathbb{T}' \backslash \{T'\} = \mathbb{T}'' \backslash \{T''\}$

   and

   $\vec{q'} \in \mathbb{QF}_{\mathbb{T}'}, \vec{q''} \in \mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'} \in \mathbb{LTF}_{\mathbb{T}'}, \vec{\tau''} \in \mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'} \in \mathbb{UF}_{\mathbb{T}'}, \vec{u''} \in \mathbb{UF}_{\mathbb{T}''}$, with

$$r'_{T'} = r''_{T''}, q'_{T'} = q''_{T''}, \tau'_{T'} = \tau''_{T''}$$

$$\forall t \in \mathbb{GT} : u'_{T'}(t) \leq u''_{T''}(t)$$

As qualities are non-negative, we know that

$$\begin{aligned}
v_{\vec{q'},\vec{u'}}(\vec{\tau'},t) &= \max_{t'\le t, T\in\mathbb{T}'} u'_T(t')\cdot q'_T(\tau'_T(t'))\\
&= \max_{t'\le t}\Big(\max\Big(u'_{T'}(t')\cdot q'_{T'}(\tau'_{T'}(t')), \max_{T\in\mathbb{T}'\setminus\{T'\}} u'_T(t')\cdot q'_T(\tau'_T(t'))\Big)\Big)\\
&= \max_{t'\le t}\Big(\max\Big(u'_{T'}(t')\cdot q''_{T''}(\tau''_{T''}(t')), \max_{T\in\mathbb{T}''\setminus\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\Big)\Big)\\
&\le \max_{t'\le t}\Big(\max\Big(u''_{T''}(t')\cdot q''_{T''}(\tau''_{T''}(t')), \max_{T\in\mathbb{T}''\setminus\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\Big)\Big)\\
&= \max_{t'\le t, T\in\mathbb{T}''} u''_T(t')\cdot q''_T(\tau''_T(t'))\\
&= v_{\vec{q''},\vec{u''}}(\vec{\tau''},t)
\end{aligned}$$

7. quality monotony

Let $\mathbb{T}',\mathbb{T}''\subseteq\mathbb{T}$ with $T'\in\mathbb{T}', T''\in\mathbb{T}''$ and $\mathbb{T}'\setminus\{T'\}=\mathbb{T}''\setminus\{T''\}$

and

$\vec{q'}\in\mathbb{QF}_{\mathbb{T}'}, \vec{q''}\in\mathbb{QF}_{\mathbb{T}''}$, $\vec{\tau'}\in\mathbb{LTF}_{\mathbb{T}'}, \vec{\tau''}\in\mathbb{LTF}_{\mathbb{T}''}$, $\vec{u'}\in\mathbb{UF}_{\mathbb{T}'}, \vec{u''}\in\mathbb{UF}_{\mathbb{T}''}$, with

$$r'_{T'}=r''_{T''}, u'_{T'}=u''_{T''}, \tau'_{T'}=\tau''_{T''}$$

$$\forall n'\in\mathbb{LT}_{T'}, n''\in\mathbb{LT}_{T''}: n'\equiv n''\Rightarrow q'_{T'}(n')\le q''_{T''}(n'')$$

As utilities are non-negative, we know that

$$\begin{aligned}
v_{\vec{q'},\vec{u'}}(\vec{\tau'},t) &= \max_{t'\le t, T\in\mathbb{T}'} u'_T(t')\cdot q'_T(\tau'_T(t'))\\
&= \max_{t'\le t}\Big(\max\Big(u'_{T'}(t')\cdot q'_{T'}(\tau'_{T'}(t')), \max_{T\in\mathbb{T}'\setminus\{T'\}} u'_T(t')\cdot q'_T(\tau'_T(t'))\Big)\Big)\\
&= \max_{t'\le t}\Big(\max\Big(u''_{T''}(t')\cdot q'_{T'}(\tau'_{T'}(t')), \max_{T\in\mathbb{T}''\setminus\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\Big)\Big)\\
&\le \max_{t'\le t}\Big(\max\Big(u''_{T''}(t')\cdot q''_{T''}(\tau''_{T''}(t')), \max_{T\in\mathbb{T}''\setminus\{T''\}} u''_T(t')\cdot q''_T(\tau''_T(t'))\Big)\Big)\\
&= \max_{t'\le t, T\in\mathbb{T}''} u''_T(t')\cdot q''_T(\tau''_T(t'))\\
&= v_{\vec{q''},\vec{u''}}(\vec{\tau''},t)
\end{aligned}$$

$\square$

B.6 Local Search Space Size

For an elementary interval $J = [t_s; t_e[$, we want to find out the number of possible assignments of the available units of processor time (equal to the length of the interval) to the tasks in this interval. We define $\nu := \left|\mathbb{T}'_J\right|$ and $\psi := t_e - t_s$.

First, we prove the following:

$$\sum_{\psi'=0}^{\psi} \binom{\psi' + \nu - 1}{\psi'} = \binom{\psi + \nu}{\psi}$$

- $\psi = 0$:

$$\sum_{\psi'=0}^{0} \binom{\psi' + \nu - 1}{\psi'} = \binom{\nu - 1}{0} = 1 = \binom{0 + \nu}{0}$$

- $\psi \to \psi + 1$:

$$\begin{aligned}
\sum_{\psi'=0}^{\psi+1} \binom{\psi' + \nu - 1}{\psi'} &= \binom{\psi + 1 + \nu - 1}{\psi + 1} + \sum_{\psi'=0}^{\psi} \binom{\psi' + \nu - 1}{\psi'} \\
&= \binom{\psi + \nu}{\psi + 1} + \binom{\psi + \nu}{\psi} \\
&= \frac{(\psi + \nu)!}{(\psi + 1)!(\nu - 1)!} + \frac{(\psi + \nu)!}{\psi!\nu!} \\
&= \frac{\nu(\psi + \nu)! + (\psi + 1)(\psi + \nu)!}{(\psi + 1)!\nu!} \\
&= \frac{(\psi + \nu + 1)(\psi + \nu)!}{(\psi + 1)!\nu!} \\
&= \frac{(\psi + 1 + \nu)!}{(\psi + 1)!\nu!} \\
&= \binom{\psi + 1 + \nu}{\psi + 1}
\end{aligned}$$

The number of possible assignment of ψ units of computation time on ν tasks with $\psi \geq 0$ and $\nu \geq 1$ is given by the recursive definition:

$$lss(\psi, 1) = 1$$

$$lss(\psi, \nu + 1) = \sum_{\psi'=0}^{\psi} lss(\psi', \nu)$$

The recursion can be resolved into

$$lss(\psi,\nu) = \binom{\psi+\nu-1}{\psi}$$

- $\nu = 1$:
$$\binom{\psi+1-1}{\psi} = \binom{\psi}{\psi} = 1 = lss(\psi,1)$$

- $\nu \rightarrow \nu+1$:

$$\begin{aligned} lss(\psi,\nu+1) &= \sum_{\psi'=0}^{\psi} lss(\psi',\nu) \\ &= \sum_{\psi'=0}^{\psi} \binom{\psi'+\nu-1}{\psi'} \\ &= \binom{\psi+\nu}{\psi} \end{aligned}$$

□

B.7 Influence of Homogeneity of Interval Lengths

Let $\nu \in \mathbb{N}_0$ and $\psi_1, \psi_2, \psi_1', \psi_2' \in \mathbb{N}_0$ with $\psi_1 + \psi_2 = \psi_1' + \psi_2'$ and $|\psi_1 - \psi_2| \leq |\psi_1' - \psi_2'|$. Then

$$lss(\psi_1,\nu) \cdot lss(\psi_2,\nu) \geq lss(\psi_1',\nu) \cdot lss(\psi_2',\nu)$$

Proof:
Without loss of generality, we assume $\psi_1 \leq \psi_2$ and $\psi_1' \leq \psi_2'$.
Define $\psi := \psi_1 + \psi_2 = \psi_1' + \psi_2'$ and $\Delta := \frac{\psi_2 - \psi_1}{2}$, $\Delta' := \frac{\psi_2' - \psi 1'}{2}$. By definition, $0 \leq \Delta \leq \Delta'$.

Therefore

$$\begin{aligned} \Delta \leq \Delta' \quad &\Rightarrow \quad \Delta^2 \leq \Delta'^2 \\ &\Rightarrow \quad \forall i \in \mathbb{N}_0 : (\frac{\psi}{2}+i)^2 - \Delta^2 \geq (\frac{\psi}{2}+i)^2 - \Delta'^2 \\ &\Rightarrow \quad \forall i \in \mathbb{N}_0 : (\frac{\psi}{2}-\Delta+i)(\frac{\psi}{2}+\Delta+i) \geq (\frac{\psi}{2}-\Delta'+i)(\frac{\psi}{2}+\Delta'+i) \\ &\Rightarrow \quad \forall i \in \mathbb{N}_0 : (\psi_1+i)(\psi_2+i) \geq (\psi_1'+i)(\psi_2'+i) \end{aligned}$$

$$\begin{aligned}
&\Rightarrow \prod_{i=1}^{\nu-1}(\psi_1+i)(\psi_2+i) \geq \prod_{i=1}^{\nu-1}(\psi_1'+i)(\psi_2'+i)\\
&\Rightarrow \prod_{i=1}^{\nu-1}(\psi_1+i)\cdot\prod_{i=1}^{\nu-1}(\psi_2+i) \geq \prod_{i=1}^{\nu-1}(\psi_1'+i)\cdot\prod_{i=1}^{\nu-1}(\psi_2'+i)\\
&\Rightarrow \frac{(\psi_1+\nu-1)!}{\psi_1!}\cdot\frac{(\psi_2+\nu-1)!}{\psi_2!} \geq \frac{(\psi_1'+\nu-1)!}{\psi_1'!}\cdot\frac{(\psi_2'+\nu-1)!}{\psi_2'!}\\
&\Rightarrow \frac{(\psi_1+\nu-1)!}{\psi_1!(\nu-1)!}\cdot\frac{(\psi_2+\nu-1)!}{\psi_2!(\nu-1)!} \geq \frac{(\psi_1'+\nu-1)!}{\psi_1'!(\nu-1)!}\cdot\frac{(\psi_2'+\nu-1)!}{\psi_2'!(\nu-1)!}\\
&\Rightarrow \binom{\psi_1+\nu-1}{\psi_1}\cdot\binom{\psi_2+\nu-1}{\psi_2} \geq \binom{\psi_1'+\nu-1}{\psi_1'}\cdot\binom{\psi_2'+\nu-1}{\psi_2'}\\
&\Rightarrow lss(\psi_1,\nu)\cdot lss(\psi_2,\nu) \geq lss(\psi_1',\nu)\cdot lss(\psi_2',\nu)
\end{aligned}$$

□

Appendix C

Components of the PaSchA Project

This chapter contains screenshots and short descriptions (mostly in tabular form) of many of the user interfaces of the PaSchA components and some other items of interest on the project in order to give the reader an overview of both their functionality and limitations. The screenshots of this chapter are not included in the list of figures of the main text.

C.1 Editor

C.1.1 Menu and Toolbar Elements

Standard elements like new/open/save/close, undo/redo or cut/copy/paste are omitted.

select from DB	load application graph from database
auto-reload graphs	switch to activate/deactivate automatic reloading of previously opened graphs when starting editor
save to DB	save application graph to database
print	print current application graph
export as image	save current application graph in bitmap format
select all	highlight all elements of current graph
unselect all	change all selected objects to unselected
insert . . . menu	choose one of the modes of operation for mouse clicks: either selecting/moving/editing elements or inserting new ones (tasks, methods, resources, hierarchy edges, dependencies)
show grid	switch to display/hide a rectangular grid on the painting pane
adjust grid	change the distance of grid lines

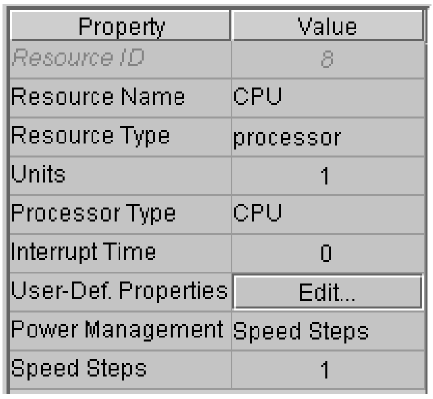
connect rect	switch to layout the hierarchy edges in rectangular fashion or as direct connections
layout	invoke an automatic layout algorithm for the graph
show edges	switch to display/hide the hierarchy edges
show dependencies	switch to display/hide the dependency edges
zoom level	select one of four zoom levels with different sizes and levels of detail for nodes
algorithm	drop-down list to select a scheduling algorithm for which this graph is intended; entries are names of subclasses of scheduler base class
param	open the parameter dialogue for the selected scheduling algorithm
check	check whether the graph complies with the correctness test for the selected scheduling algorithm together with the algorithm-specific parameters

C.1.2 Object-Specific Elements

The following attributes are common to all node types:

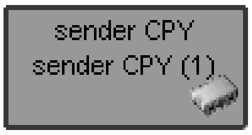
ID	unique, not editable identifier
name	character string description, only relevant to human application designer

C.1.2.1 Processors



Property	Value
Resource ID	*8*
Resource Name	CPU
Resource Type	processor
Units	1
Processor Type	CPU
Interrupt Time	0
User-Def. Properties	Edit...
Power Management	Speed Steps
Speed Steps	1

dialogue

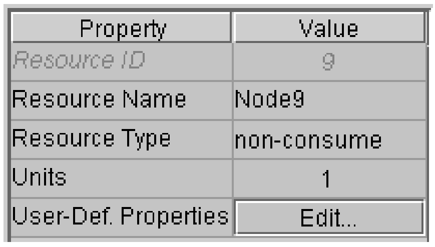


pictogram

resource type	invariable, equals *processor*
units	number of processors of this type available

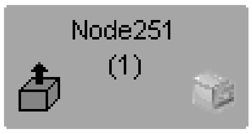
processor type	unique character string description to be referenced by methods
interrupt time	context switch costs, defaults to 0
user-defined properties	optional list of additional properties
power management	drop-down list of available power management models
speed steps	available speed steps for this processor

C.1.2.2 Resources

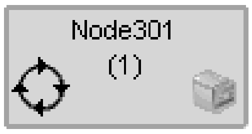


Property	Value
Resource ID	9
Resource Name	Node9
Resource Type	non-consume
Units	1
User-Def. Properties	Edit...

dialogue



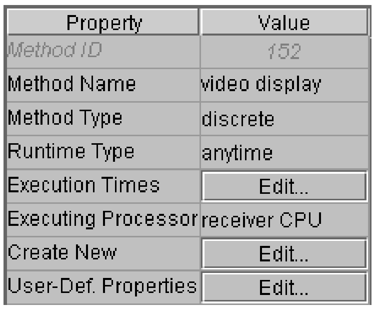
consumable resource



non-consumable resource

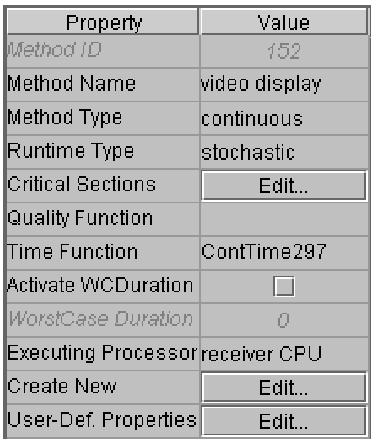
resource type	either *consumable* or *non-consumable*
units	number of units of this resource available
user-defined properties	optional list of additional properties

C.1.2.3 Methods



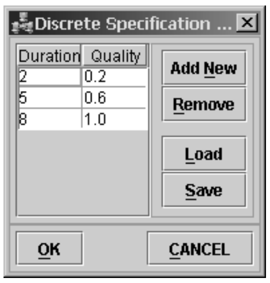
Property	Value
Method ID	152
Method Name	video display
Method Type	discrete
Runtime Type	anytime
Execution Times	Edit...
Executing Processor	receiver CPU
Create New	Edit...
User-Def. Properties	Edit...

main dialogue for anytime method with discrete quality function



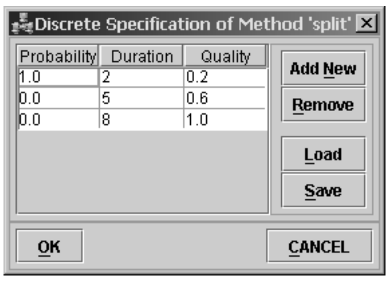
Property	Value
Method ID	152
Method Name	video display
Method Type	continuous
Runtime Type	stochastic
Critical Sections	Edit...
Quality Function	
Time Function	ContTime297
Activate WCDuration	☐
WorstCase Duration	0
Executing Processor	receiver CPU
Create New	Edit...
User-Def. Properties	Edit...

main dialogue for stochastic (run-to-completion) method with continuous probability distribution function



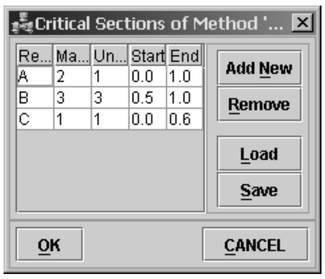
Duration	Quality
2	0.2
5	0.6
8	1.0

quality function of anytime method



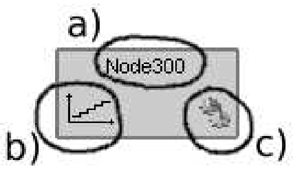
Probability	Duration	Quality
1.0	2	0.2
0.0	5	0.6
0.0	8	1.0

probability distribution of stochastic method

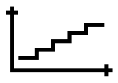


Re...	Ma...	Un...	Start	End
A	2	1	0.0	1.0
B	3	3	0.5	1.0
C	1	1	0.0	0.6

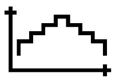
critical sections dialogue



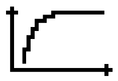
Pictogram: a) name, b) method and execution type, c) general method symbol



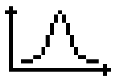
discrete, anytime



discrete, run-to-completion



continuous, anytime

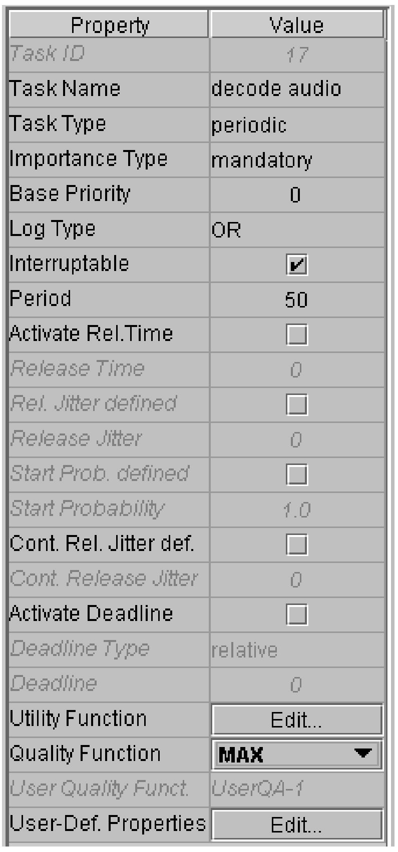


continuous, run-to-completion

method type	either *discrete* or *continuous*; refers to the type of specification of the quality or distribution function
runtime type	either *anytime* or *stochastic*; refers to the execution paradigm: execution time determined by the scheduler (*anytime*) or by the simulator according to a probability distribution (*stochastic*)
execution times	opens dialogue to enter discrete specification of quality function (runtime type *anytime*) or probability distribution (runtime type *stochastic*); only for method type *discrete*
critical sections	opens dialogue to edit list of critical sections; only for runtime type *stochastic*
worst-case duration	worst-case execution time of method together with a switch for activation/deactivation; only for runtime type *stochastic*
quality function	name of Java method implementing the quality function (mapping execution time to quality); only for method type *continuous*

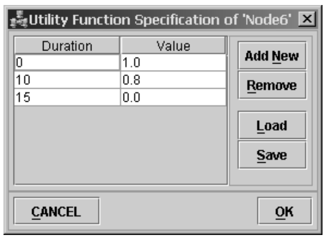
time function	name of Java method implementing the probability distribution for execution times; only for method type *continuous* and run-time type *stochastic*
executing processor	processor type on which this method is executable
create new	opens dialogue to create new processor type
user-defined properties	optional list of additional properties

C.1.2.4 Tasks



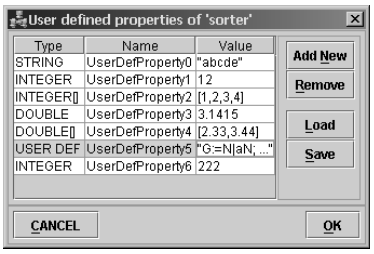
Property	Value
Task ID	17
Task Name	decode audio
Task Type	periodic
Importance Type	mandatory
Base Priority	0
Log Type	OR
Interruptable	✓
Period	50
Activate Rel.Time	☐
Release Time	0
Rel. Jitter defined	☐
Release Jitter	0
Start Prob. defined	☐
Start Probability	1.0
Cont. Rel. Jitter def.	☐
Cont. Release Jitter	0
Activate Deadline	☐
Deadline Type	relative
Deadline	0
Utility Function	Edit...
Quality Function	MAX
User Quality Funct.	UserQA-1
User-Def. Properties	Edit...

main dialogue



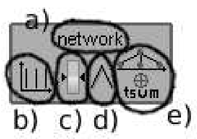
Duration	Value
0	1.0
10	0.8
15	0.0

utility function dialogue



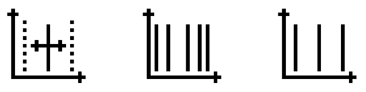
Type	Name	Value
STRING	UserDefProperty0	"abcde"
INTEGER	UserDefProperty1	12
INTEGER[]	UserDefProperty2	[1,2,3,4]
DOUBLE	UserDefProperty3	3.1415
DOUBLE[]	UserDefProperty4	[2.33,3.44]
USER DEF	UserDefProperty5	"G:=N|aN; ..."
INTEGER	UserDefProperty6	222

user-defined properties dialogue



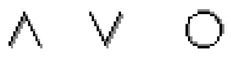
pictogram: a) name b) instantiation type c) importance d) logical type e) quality function

instantiation types:



sporadic aperiodic periodic

logical types:



and or atomic

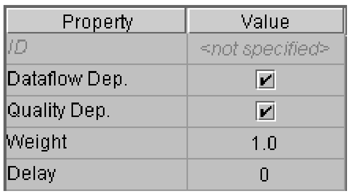
quality functions:

instance-based minimum	instance-based maximum	instance-based sum	instance-based average	user-defined
time-based minimum	time-based maximum	time-based sum	time-based average	density

task type	either *sporadic*, *periodic*, or *aperiodic*
importance type	either *mandatory*, *high*, *medium*, *low*, or *background*
base priority	integer value for initial priority
log type	either *and* or *or*
interruptable	switch for preemptive/nonpreemptive tasks
release time	release time of task together with a switch for activation/deactivation (inactive meaning release time of 0)
rel. jitter	the interval width for a uniform distribution of the release time of the first instance together with a switch for activation/deactivation
start probability	probability for a geometric distribution of release time of the first instance together with a switch for activation/deactivation
deadline	the value of the deadline specification together with a switch for activation/deactivation
deadline type	either *relative* to the release time or *absolute* (in fact, relative to the parent node's release time)
utility function	opens dialogue to edit the discrete utility function for this task
user-def. properties	set of optional user-defined properties
period	period length (only for *periodic* tasks)
cont. release jitter	jitter value for second and subsequent instances together with a switch for activation/deactivation (only for *periodic* tasks)
min. interarrival time	minimum interarrival time between instances (only for *aperiodic* tasks)

interarrival prob	release probability for second and subsequent instances together with a switch for activation/deactivation (only for *aperiodic* tasks)
quality function	defining the value of an instance of this node as one of: • sum of the corresponding child instance values • maximum of the corresponding child instance values • minimum of the corresponding child instance values • arithm. mean of the corresponding child instance values • sum of all prior child instance values • maximum of all prior child instance values • minimum of all prior child instance values • arithmetic mean of all prior child instance values • average value gain of child instances per time unit • calculated according to a user-provided method
user quality function	name of Java method implementing the user-defined quality function

C.1.2.5 Dependency Edges



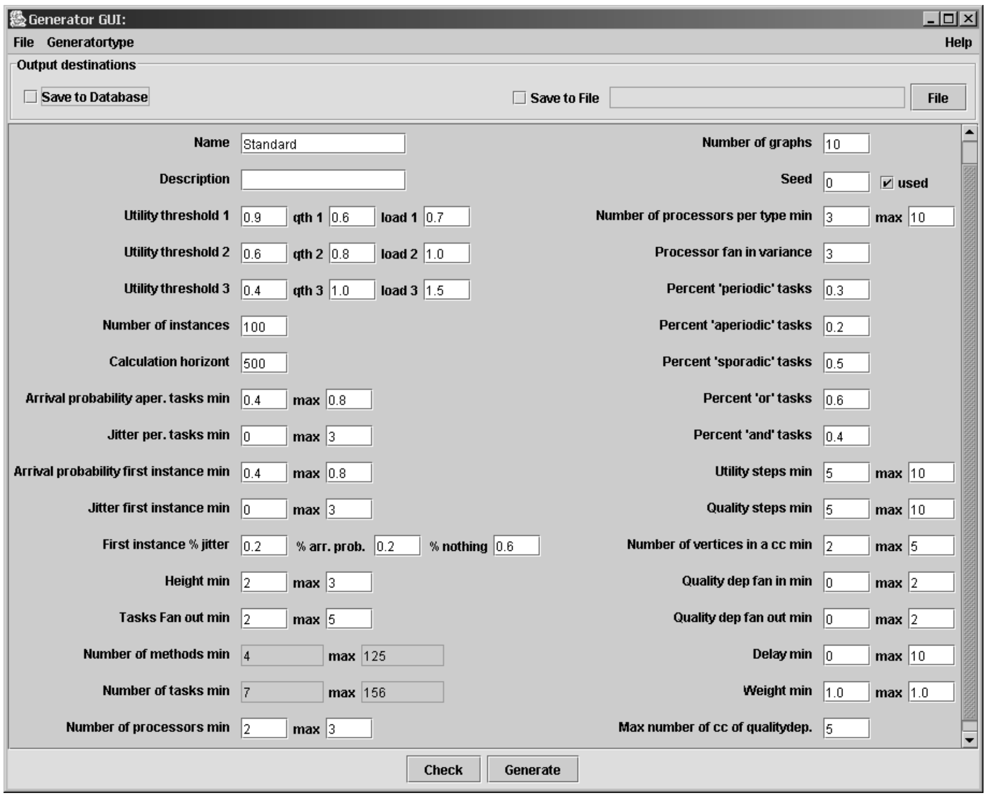
Property	Value
ID	*<not specified>*
Dataflow Dep.	✓
Quality Dep.	✓
Weight	1.0
Delay	0

dialogue

dataflow dep.	if selected, this edge describes a precedence constraint; the successor node must not start before the predecessor has finished
quality dep.	if selected, this edge describes a value (quality) dependency; the value of the predecessor node influences the value of the successor node
weight	the level of influence of the predecessor on the successor node in a value dependency
delay	the distance in instance numbers of nodes depending on each other

C.2 Graph Generator

Three graph generators were developed to provide generic loads for a static scheduling and power management problem, to produce graphs in the TGFF format, and to generate test graphs for the quality / utility scheduling problem of this work; only the latter one will be described here.



☐ Save to Database ☐ Save to File

Name	Standard						Number of graphs	10		
Description							Seed	0	✓ used	
Utility threshold 1	0.9	qth 1	0.6	load 1	0.7		Number of processors per type min	3	max	10
Utility threshold 2	0.6	qth 2	0.8	load 2	1.0		Processor fan in variance	3		
Utility threshold 3	0.4	qth 3	1.0	load 3	1.5		Percent 'periodic' tasks	0.3		
Number of instances	100						Percent 'aperiodic' tasks	0.2		
Calculation horizont	500						Percent 'sporadic' tasks	0.5		
Arrival probability aper. tasks min	0.4	max	0.8				Percent 'or' tasks	0.6		
Jitter per. tasks min	0	max	3				Percent 'and' tasks	0.4		
Arrival probability first instance min	0.4	max	0.8				Utility steps min	5	max	10
Jitter first instance min	0	max	3				Quality steps min	5	max	10
First instance % jitter	0.2	% arr. prob.	0.2	% nothing	0.6		Number of vertices in a cc min	2	max	5
Height min	2	max	3				Quality dep fan in min	0	max	2
Tasks Fan out min	2	max	5				Quality dep fan out min	0	max	2
Number of methods min	4	max	125				Delay min	0	max	10
Number of tasks min	7	max	156				Weight min	1.0	max	1.0
Number of processors min	2	max	3				Max number of cc of qualitydep.	5		

Graph generator for quality/utility graphs

C.2.1 Menu and Toolbar Items

The standard items are omitted.

save to database	switch to enable/disable writing of graphs to database
save to file	switch to enable/disable writing of graphs to file system
check	test whether the parameters are consistent
generate	generate a set of graphs and write them to the database and/or file system

C.2.2 Input Parameters for Generator Algorithm

name	name of a parameter set
description	additional comment on the parameter set
utility threshold 1	the utility threshold for the first of three tuples of parameters to direct the utilisation incurred by the graphs to generate (see below)
qth 1	the quality threshold for the first of three tuples of parameters to direct the utilisation incurred by the graphs to generate (see below)
load 1	in the quality / utility scheduling model, utilisation is defined on the basis of quality thresholds (to determine minimum execution times) and utility thresholds (to define minimum utility deadlines); test graphs should exhibit a utilisation of approximately *load 1* if the corresponding thresholds (the two preceding parameters) are applied; to this end, utility and quality functions in the graph have to be modified simultaneously; with a complex definition of load, a large number of tasks and methods with irregular AND/OR hierarchy trees and several threshold/load tuples, this is difficult to do by hand
utility threshold 2/3	two further utility thresholds
qth 2/3	two further quality thresholds
load 2/3	see explanation for *load 1*
number of instances	number of task instances to be generated from a graph within the calculation horizon (determining the period lengths, minimum interarrival times, and release probabilities)
calculation horizon	size of time window into the future; basis for assessing the number of instances
arrival probability aper. tasks min/max	minimum/maximum value for the arrival probability of aperiodic tasks
jitter per. tasks min/max	minimum/maximum jitter value for periodic tasks
arrival probability first instance min/max	minimum/maximum value for the arrival probability of first instance of tasks
jitter first instance min/max	minimum/maximum jitter value of first instance of tasks

probability first instance jitter	approximate percentage of tasks having a jitter specification (uniform distribution) for their first instance; only either jitter, arrival probability, or none of these can be specified
arr. prob.	approximate percentage of tasks having an arrival probability specification (geometric distribution) for their first instance of a task
directly start	approximate percentage of tasks having neither jitter nor release probability specifications for their first instances, so that the release time is given deterministically
height min/max	minimum/maximum height of hierarchy graph
tasks fan out min/max	minimum/maximum fan-out value of non-leaf nodes in hierarchy graph
number of methods min/max	minimum/maximum number of methods in graph
number of tasks min/max	minimum/maximum number of tasks in graph
number of processors min/max	minimum/maximum number of processor types in graph
number of graphs	number of graphs to be generated
seed	random seed to make results reproducible
used	switch to turn on/off random seed usage
number of processors per type min/max	minimum/maximum number of instances of processor type
processor fan in variance	variance of number of methods executable on each processor type; influences how evenly distributed the load among processors is
probability ’periodic’ tasks	approximate percentage of leaf task nodes being periodic or direct or indirect children of periodic tasks
probability ’aperiodic’ tasks	approximate percentage of leaf task nodes being aperiodic or direct or indirect children of aperiodic tasks
probability ’sporadic’ tasks	approximate percentage of leaf task nodes being sporadic and not direct or indirect children of any periodic or aperiodic tasks
probability ’or’ tasks	approximate percentage of inner task nodes being of logical type *or*
probability ’and’ tasks	approximate percentage of inner task nodes being of logical type *and*
utility steps min/max	minimum/maximum number of steps in utility functions

quality steps min/max	minimum/maximum number of steps in quality functions
number of vertices in a cc min/max	minimum/maximum number of vertices in a connective component (maximum contiguous subgraph) of the dependency graph
quality dep fan in min/max	minimum/maximum fan-in value of non-source nodes in dependency graph
quality dep fan out min/max	minimum/maximum fan-out value of non-sink nodes in dependency graph
delay min/max	minimum/maximum delay value for dependency edges
weight min/max	minimum/maximum weight for dependency edges
max number of cc of qualitydep.	maximum number of connective components of the dependency graph

C.3 Simulator

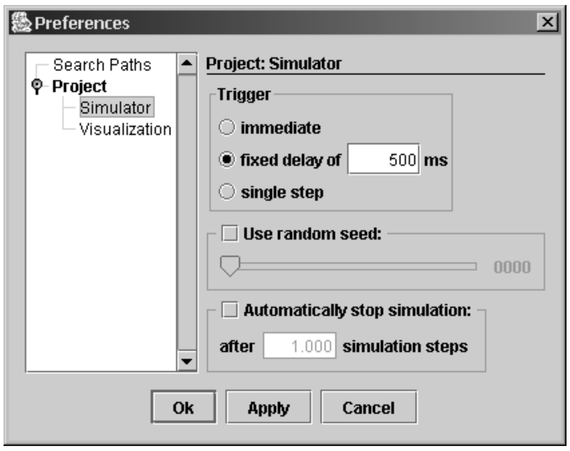
C.3.1 Graphical User Interface – Menu and Toolbar Items

Standard items are omitted.

properties	open the configuration editor for the selected configuration; if no configuration has been selected, open the file system browser first
new config	create a new configuration and open the configuration editor for it
add config	add a configuration or a log file to the play list
remove	remove a configuration or a log file from the play list
preferences	open the preferences dialogue
up arrow	move selected configuration or log file upward in play list
down arrow	move selected configuration or log file downward in play list
init	initialise all configurations and log files in the play list (must not have previously been initialised)
init+start	initialise and start all configurations and log files in the play list (must not have previously been initialised)
start	start all configurations and log files in the play list (must have previously been initialised)

stop	stop all running simulation and log playing threads
cleanup	remove all simulation and log playing threads and close all visualisation windows
three state buttons	displayed separately for each configuration and log file to alter the visualisation preferences; states: grey: use defaults for the visualisation mode defined in preferences, green: override default settings to force display of the visualisation mode, red: override default settings to force hiding of the visualisation mode; the four buttons refer to the time view (TV), graph view (GV), log view (LV), and statistics view (SV), respectively

C.3.2 Graphical User Interface – Preferences



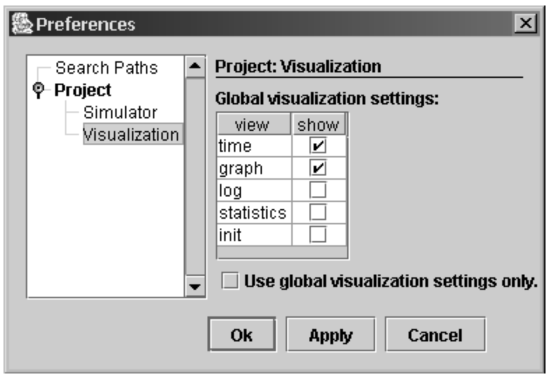
Project: Simulator

Trigger
- immediate
- (●) fixed delay of 500 ms
- single step

☐ Use random seed: 0000

☐ Automatically stop simulation: after 1.000 simulation steps

simulator preferences dialogue



Project: Visualization

Global visualization settings:

view	show
time	✓
graph	✓
log	☐
statistics	☐
init	☐

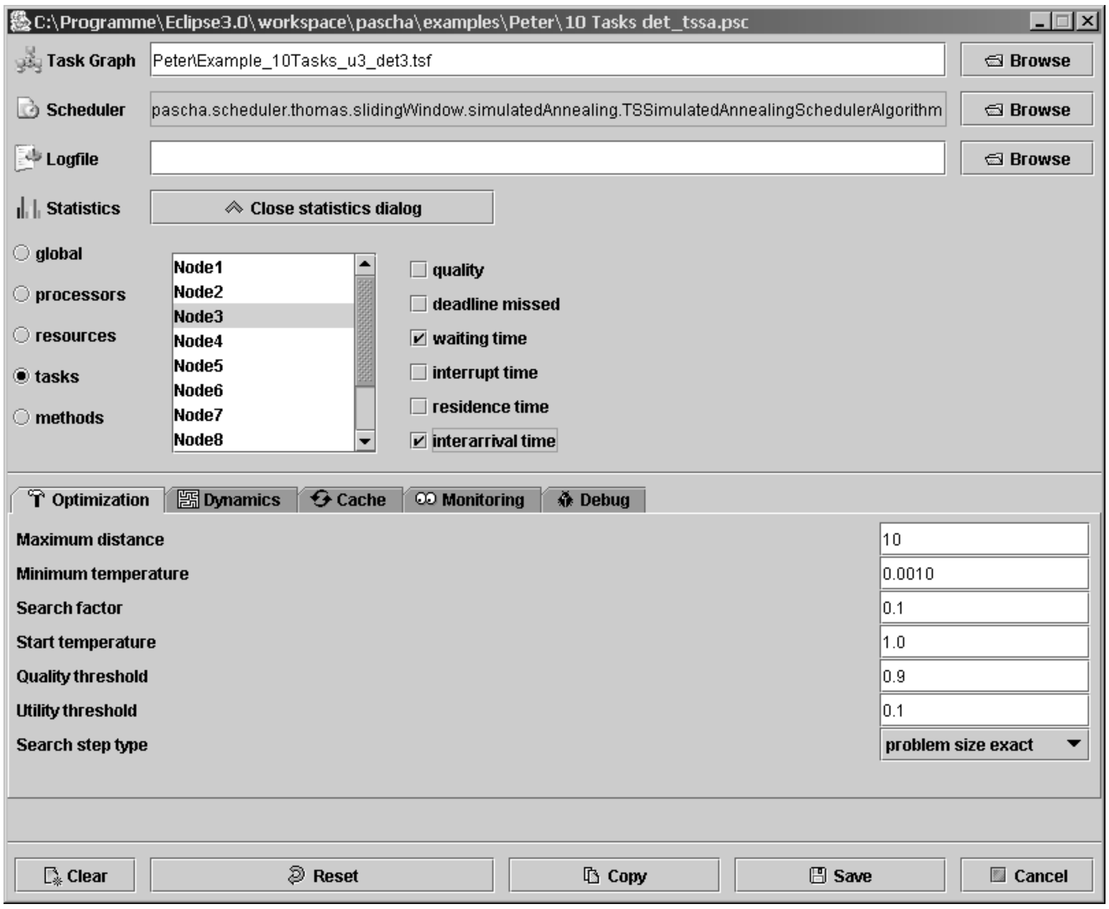
☐ Use global visualization settings only.

visualisation preferences dialogue

search paths	list of absolute file system paths from which to start the search for relative paths
trigger type	either *immediate* (simulate as fast as possible), with a *fixed delay* between steps, or in *single-step mode* with user interaction
ms	duration in ms for fixed delay
use random seed	switch to turn on/off usage of a random seed for reproducible results
random seed slider	select a value for the random seed

automatically stop simulation after . . . steps	switch to turn on/off automatic stopping and input field for number of steps
time view show	switch to turn on/off time view visualisation mode by default
graph view show	switch to turn on/off graph view visualisation mode by default
log view show	switch to turn on/off log view visualisation mode by default
statistics view show	switch to turn on/off statistics view visualisation mode by default
use global visualization settings only	allow/disallow overriding of default settings for visualisation

C.3.3 Configuration Editor



C:\Programme\Eclipse3.0\workspace\pascha\examples\Peter\10 Tasks det_tssa.psc

Task Graph	Peter\Example_10Tasks_u3_det3.tsf
Scheduler	pascha.scheduler.thomas.slidingWindow.simulatedAnnealing.TSSimulatedAnnealingSchedulerAlgorithm
Logfile	
Statistics	

○ global
○ processors
○ resources
● tasks
○ methods

Node1
Node2
Node3
Node4
Node5
Node6
Node7
Node8

☐ quality
☐ deadline missed
☑ waiting time
☐ interrupt time
☐ residence time
☑ interarrival time

Maximum distance	10
Minimum temperature	0.0010
Search factor	0.1
Start temperature	1.0
Quality threshold	0.9
Utility threshold	0.1
Search step type	problem size exact

Configuration editor with simulated-annealing parameter dialogue (lower half)

task graph	name of application graph file
scheduler	name of a scheduler class
logfile	optional name of a log file to write simulation results to
statistics	button to show/hide statistics parameter section
global	non-object-related statistics data: number of ready tasks, number of ready methods, number of working tasks, number of working methods, time consumption of a scheduling step, number of deadlines missed, number of working processors
processors	shows the list of processor types of the selected application graph; statistics item for each processor type: utilisation
resources	shows the list of resources of the selected application graph; statistics item for each resource: usage
tasks	shows the list of tasks of the selected application graph; statistics items for each task: quality, deadline missed, waiting time, interrupt time, residence time, interarrival time
methods	shows the list of methods of the selected application graph; statistics items for each method: quality, waiting time, interrupt time, residence time
scheduler-specific parameter section	the lower part of the configuration editor contains a dialogue which can optionally be provided by the scheduler class to allow setting of the parameters for the scheduling algorithm

C.3.4 Simulator Main – Simulation Cycle

A high-level description of the simulator main loop is shown below.

The states of task and method instances within a hierarchy are not independent of each other. For example, an instance of an OR type task is considered active if one or more of its children are active, and an instance of an AND type task is only considered finished if all of its children are finished. Therefore, it is necessary to regularly calculate new states for some nodes in the instance graph starting from those leaves where changes have appeared. These state changes are performed in the function `bottomUpPropagateStatus`. The next action in the first phase of the simulator loop is to determine the set of finished run-to-completion type methods, followed by the creation of new instances of tasks and methods, if necessary. Note that in general these instances are not released (i.e., made known to the outside world) at the same point in time, but are created ahead of the prospective release time to cope with jitter and data dependency effects. The following steps are the release of instances (i.e., making them known to the outside world and ready for execution) and checking whether methods enter or leave some critical section and whether all precedence constraints can be met. The final actions prior to invoking the scheduler are the detection of missed deadlines and removing finished instances.

After the scheduler has run, the simulator has to react to its actions by propagating state and quality changes in bottom-up manner starting from the leaves and removing any nodes the scheduler may have decided to terminate.

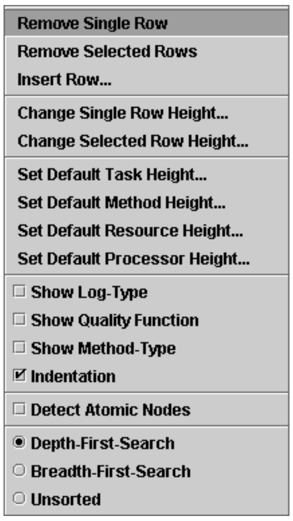
```
procedure simulator;
  var schedulingHorizon;
  begin $time := 0$; $state := startState$;
    while $error \neq true$ and $time < schedulingHorizon$ do
      begin
        bottomUpPropagateStatus(state, time); // parent state defined over child states
        determineFinishedMethods(state, time); // for run-to-completion methods
        createNewNodeInstances(state, time); // well ahead of release time
        releaseNodes(state, time); // once release time has arrived
        checkForCriticalSections(state, time);
        activateOrBlock(state, time); // according to critical sections and resource allocation
        checkDeadlines(state, time);
        detectFinishedNodes(state, time); // do not yet remove; may need data later
        removeOldNodes(state, time); // if data are no longer needed

        executeScheduler(state, time); // pass on control to the scheduler

        bottomUpPropagateStatus(state, time);
        bottomUpPropagateQuality(state, time);
        detectFinishedNodes(state, time); // for anytime methods
        removeOldNodes(state, time);
        $time := time + 1$;
    end
end
```

Simulator main loop

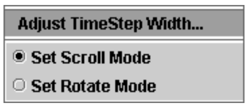
C.4 Visualisation

C.4.1 Time View

At run-time, the time view visualisation mode has several options, which can be accessed via the context menus of the name tag window to the left and the painting area to the right:



name tag window context menu



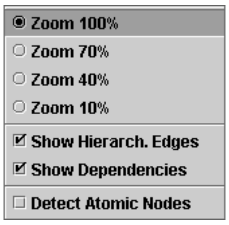
painting area context menu

remove single row	hide entry for one object (task, method, etc.)
remove selected rows	hide entries for set of objects
insert row	insert entry for object
change single row height	change the height of one entry
change selected row height	change the height of several entries
set default task height	change default height for all task entries
set default method height	change default height for all method entries
set default resource height	change default height for all resource entries
set default processor height	change default height for all processor entries
show log-type	show/hide logical types of tasks (AND/OR) in the name tag

show quality function	show/hide quality function of tasks (sum,tsum,max,tmax,. . .) in the name tag
show method-type	show/hide method type (discrete anytime, discrete stochastic, continuous anytime, continuous stochastic) in the name tag
indentation	switch on/off indentation of elements according to position in hierarchy
detect atomic nodes	turn on/off display of logical type ATOMIC for leaf nodes
depth-first-search	sort entries according to depth-first search
breadth-first search	sort entries according to breadth-first search
unsorted	sort entries manually by drag-and-drop
adjust time step width	change width of unit-time rectangles for all entries in the painting area
set scroll mode	when the painting area is full, scroll it to the left to accommodate new time steps
set rotate mode	when the painting area is full, gradually overwrite its contents starting from the left

Additionally, when selecting a rectangle in the painting area, this rectangle is magnified to fit the painting area.

C.4.2 Graph View

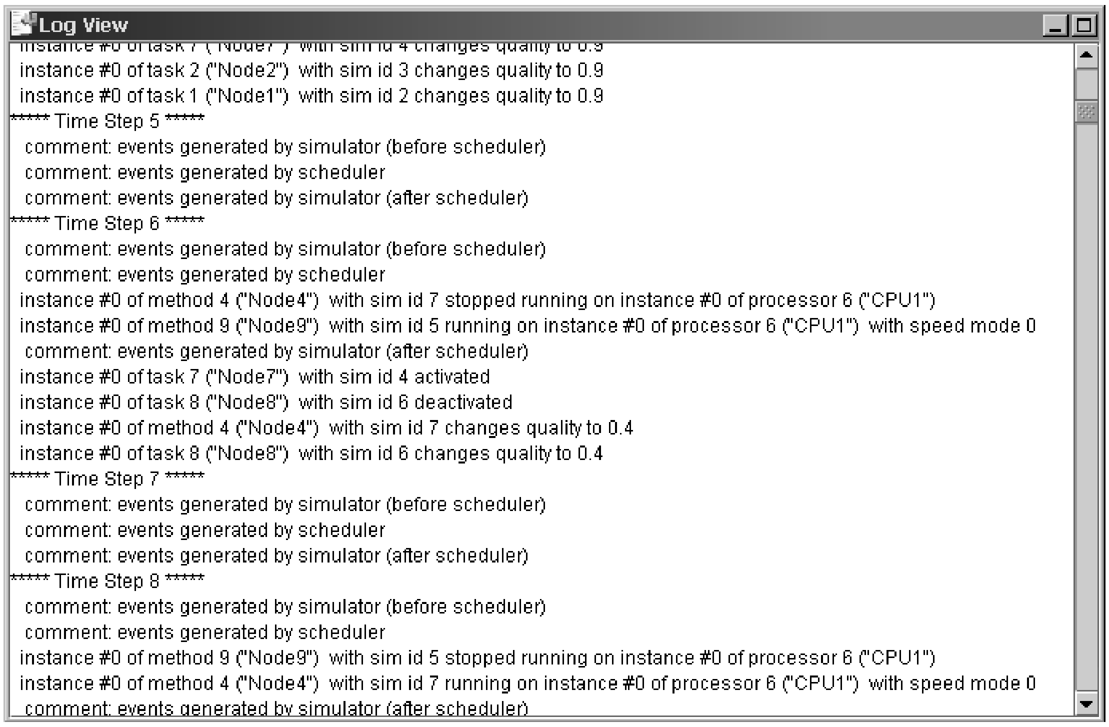


Graph view context menu

zoom 100%	nodes are shown largest with biggest set of properties
zoom 70%	nodes are shown second largest with second biggest set of properties
zoom 40%	nodes are shown second smallest with second smallest set of properties
zoom 10%	nodes are shown smallest with smallest set of properties
show hierarch. edges	show/hide hierarchy edges
show dependencies	show/hide dependency edges
detect atomic nodes	turn on/off display of logical type ATOMIC for leaf nodes

C.4.3 Log View

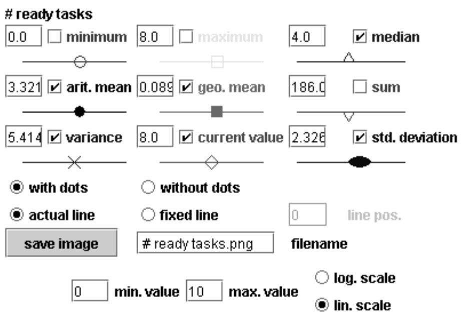
A sample output of the log view window is shown below; for each time step, three sets of events are generated: one by the simulator before invoking the scheduler, one by the scheduler, and one by the simulator after return of control from the scheduler.



```
 instance #0 of task 7 ("Node7")  with sim id 4 changes quality to 0.9
 instance #0 of task 2 ("Node2")  with sim id 3 changes quality to 0.9
 instance #0 of task 1 ("Node1")  with sim id 2 changes quality to 0.9
***** Time Step 5 *****
  comment: events generated by simulator (before scheduler)
  comment: events generated by scheduler
  comment: events generated by simulator (after scheduler)
***** Time Step 6 *****
  comment: events generated by simulator (before scheduler)
  comment: events generated by scheduler
 instance #0 of method 4 ("Node4")  with sim id 7 stopped running on instance #0 of processor 6 ("CPU1")
 instance #0 of method 9 ("Node9")  with sim id 5 running on instance #0 of processor 6 ("CPU1")  with speed mode 0
  comment: events generated by simulator (after scheduler)
 instance #0 of task 7 ("Node7")  with sim id 4 activated
 instance #0 of task 8 ("Node8")  with sim id 6 deactivated
 instance #0 of method 4 ("Node4")  with sim id 7 changes quality to 0.4
 instance #0 of task 8 ("Node8")  with sim id 6 changes quality to 0.4
***** Time Step 7 *****
  comment: events generated by simulator (before scheduler)
  comment: events generated by scheduler
  comment: events generated by simulator (after scheduler)
***** Time Step 8 *****
  comment: events generated by simulator (before scheduler)
  comment: events generated by scheduler
 instance #0 of method 9 ("Node9")  with sim id 5 stopped running on instance #0 of processor 6 ("CPU1")
 instance #0 of method 4 ("Node4")  with sim id 7 running on instance #0 of processor 6 ("CPU1")  with speed mode 0
  comment: events generated by simulator (after scheduler)
```

Log view mode

C.4.4 Statistics View



ready tasks

0.0	☐ minimum	8.0	☐ maximum	4.0	✓ median
3.321	✓ arit. mean	0.089	✓ geo. mean	186.0	☐ sum
5.414	✓ variance	8.0	✓ current value	2.326	✓ std. deviation

[✓ with dots] without dots

[✓ actual line] fixed line 0 line pos.

filename: # ready tasks.png

min. value: 0 max. value: 10 log. scale [✓ lin. scale]

Statistics view options

checkboxes	selection of graphs to plot; for each object-related or non-object-related statistics item registered to be monitored in the configuration editor, several pieces of data can be generated and displayed; these are: • current value • minimum • maximum • median • sum • arithmetic mean • geometric mean • standard deviation • variance
with dots / without dots	add / hide symbols to the colour-coded graphs; additional symbols are convenient for monochrome media
actual line / fixed line (+ line pos.)	switch between a vertical line being displayed at the current or some fixed time
save image + file name	save a bitmap representation of the current graphs to the file system under the given name
min. / max. value	minimum and maximum values for the graphs
log. / lin. scale	switch between linear and logarithmic scale

C.5 Web Site and References

Online information on the PaSchA project along with download instructions can be found at the following address:
`http://lrs.fmi.uni-passau.de/˜pascha`

The application model was introduced in [Ehr02]. Implementation details can be found for the graphical user interfaces in [Jün01], for graph generators in [Wei04], and for the benchmark components in [Fli04]. Certain aspects of the visualisation and simulator components are addressed in [Sch02, Lim04, Luc04, Mül04]. The CPLEX interface is described in [Zim04], and the adaptation of various scheduling algorithms for PaSchA in [Dem02, Sch04a, Zac04, Bus04].