Supporting Deliberative Real-Time AI Systems: A Fixed Priority Scheduling Approach

Yanching Chu

Real-time Systems Research Group
Department of Computer Science
University of York, UK

Feb 2013

A thesis submitted for the degree of Master of Philosophy.

Abstract

Constructing deliberative real-time AI systems is challenging due to the high execution-time variance in AI algorithms and the requirement of worst-case bounds for hard real-time guarantees, often resulting in poor use of system resources. Using a motivating case study based on RoboCup, the general problem of resource usage maximization in a real-time AI agent is addressed.

In this thesis it was shown that by employing a hybrid task model, for imprecise computation of the form “Prologue-Optional-Epilogue”, a variety of AI algorithms with different hard and optional (anytime) timing requirements can be supported. An exact schedulability, based on a simple offset between the Prologue and Epilogue components, is devised but shown to be computationally prohibitive when a large number of imprecise tasks are present. For this reason, a tractable sufficient schedulability test was also devised, inspired by Tindell’s analysis, where the time complexity of calculating the busy period of each task is reduced from $O(2^n)$ to $O(2n)$.

Further, with a novel scheduling scheme based on Dual Priority Scheduling, schedulability can be guaranteed for the hard Prologue and Epilogue tasks while the latter can be delayed as much as possible for allowing optional and anytime components to be executed for enhancing system utility. Suggestions on how aperiodic tasks can be scheduled effectively within the framework and how tasks can be prioritized based on their utilities by an efficient algorithm are also provided.

The works presented in this thesis provide new advances in fixed priority response time analysis for imprecise computation which, together with present results in the literature, should provide a more comprehensive scheduling framework where real-time AI systems can be suitably supported.

Contents

Acknowledgements

I am forever indebted to Prof. Alan Burns for his guidance and help as my supervisor. I thank Prof. Andy Wellings for his general support and guidance during my study, particularly as my internal examiner for this thesis. They are both like a grandfather figure in the research group taking good care of everyone.

It is a pleasure to acknowledge Dr. Rob Davis for his helpful comments and discussions, especially on drafts of various technical reports I published in the department.

Discussions with Dr. Neil Audsley on his work on scheduling imprecise computation have been extremely helpful.

I am grateful to Judith Pink in the department for her help in the submission, printing and binding of my thesis.

The research is sponsored, in part, by the EU FRESCOR project and Major International Scholarship from the University of York.

Declaration

This thesis contains original works by the author only, unless stated otherwise explicitly in the text.

Parts of Chapters 4 and 5 were published as departmental technical report 402 [31]. Parts of Chapter 6 were published as departmental technical report 414 [32]. Parts of Chapter 7 were published as departmental technical report 424 [34].

Parts of Chapters 4 and 5 were published in the proceedings of the New Trends in Real-Time Artificial Intelligence Workshop within the European Conference in Artificial Intelligence in 2006 [33].

Parts of Chapters 4-6 were published in the conference proceedings of Euromicro Conference on Real-Time Systems (ECRTS) in 2007 [35].

Parts of Chapters 4-7 were published in the Real-Time Systems Journal in 2008, in a special edition for selected papers from ECRTS 2007 [36].

Yanching Chu
Feb 2013

Chapter 1

Introduction

Real-time systems are ones for which the correctness of the system depends not only on the correct computation but also on the times that they are delivered [107, 28]. Late computation or reaction of a system may lead to catastrophic consequences due to the fact that they have been applied in domains like chemical and nuclear plant control, military operations, and railway switching systems. Predictability is a central issue concerned in real-time computing [110].

Real-time systems are becoming more complex as more intelligent behaviours are required [108, 81]. Examples are intensive care unit monitoring, robotics, avionics, space missions and military applications. As a result more AI techniques are considered to be employed in real-time systems.

Artificial Intelligence (AI) research, traditionally, has not been concerned about real-time performance of systems [88]. Many problem-solving algorithms in AI, such as planning and decision making with a knowledge base, make extensive use of searching through a large solution space whose running times are unpredictable. Even worse, the time available for some particular computation may vary in a *dynamic* environment. When advanced AI technologies are desirable in real-time domains, the unpredictability can become problematic.

Nevertheless, to exhibit intelligent behaviours, *deliberations*, which involve planning, reasoning, making decisions and predicting their effects in order to act, are necessary. The kind of intelligent AI systems which requires deliberations are called Deliberative Agents [96]. Because deliberations are computationally expensive, various techniques have been developed to make AI algorithms more real-time. The overlapping subarea of these two fields of research is termed Real-Time AI (RTAI)[1].

[1]A less commonly used name is real-time intelligent control.

One approach to the problem of unpredictable execution times of AI algorithms is to use solutions that are "good enough" rather than ones that are complete. This is referred as "satisficing" in the seminal work on *Bounded Rationality* by Herbert Simon [105]. A satisficing solution is an approximation to the exact solution. There are broadly two different kinds of approximation algorithms - iterative refinement and multiple methods, both allow trading off solution quality for computation time. Depending on the timing constraints the best possible approximation is used, the duration of computation is thus made predictable.

Iterative refinement, perhaps more commonly known as Anytime Algorithms [44], has the advantage that it can be stopped and have an answer at hand anytime. The reason why anytime algorithms are important is that while an AI algorithm may take a long time to generate complete results, they often can generate very good partial results in a much shorter time. Multiple Methods [75] share a similar spirit in that a different method can be used when time is not enough, and therefore providing quality tradeoffs.

As Russell and Wefald pointed out, flexible, autonomous systems in complex environments require the ability to reason about the appropriate resources to allocate to computation at any point, and about which computations will be most effective [93]. Here AI systems which utilize this kind of technique to compromise when full deliberation is impossible in real-time environments are considered, in order to provide a smoother transition from pure reacting to planning. And they are referred to as *Deliberative Agents* in the rest of this thesis and we assume that they have to operate in real-time and perform tradeoffs when needed.

Being able to reason about the time constraints and to decide how much time to use for obtaining good enough results can make AI systems more adaptive. This is an example of Bounded Rationality because agents need to perform rationally under "recognised" resource constraints, such as available time and computation resources enabled by the underlying architecture. Anytime algorithms have been used as a means of realizing bounded rationality in agents [124], where the system is composed of a set of anytime algorithms. Moreover, meta-level reasoning is necessary when one wants to incorporate the available reasoning resources, such as processing space and time, into the reasoning process itself. Therefore, optimal partitions of meta-reasoning and actual reasoning [61] has been investigated. Decision theory has been applied in deliberation scheduling for dealing with uncertain information and outcomes. Furthermore, deciding how much time to allocate for deliberation and acting where the deliberation scheduling can itself be an anytime algorithm [22].

However, RTAI research has been focusing on time-constrained techniques (e.g., anytime algorithm) without concerning issues normally present in real-time systems. For example, in a real-time system, in order to guarantee a set of tasks to be schedulable, their worst case execution time (WCET), arrival times and deadlines must be known a priori. At run time, however, tasks may not execute up to their WCET and not suffered maximum interference by all the eligible higher priority tasks. Slack is thus available which may be used to facilitate longer execution of anytime algorithm to yield higher quality results. Techniques like milestone methods, dynamic slack reclaiming, multiple servers and capacity sharing, as reviewed in the next two chapters, are potentially applicable in intelligent real-time agents which need to tradeoff computation quality against time.

1.1 Motivation

One observation is that as AI is concerning more about real-time constraints and being used in safety-critical hard real-time domains, increasingly it has to consider and possibly re-use various technologies that have already been developed in real-time systems research. For example, resource access and sharing are likely to be required by more complex anytime algorithms in the future. A resource access protocol that guarantees bounded blocking time is important because a high priority task can be blocked by a low priority task with an unknown amount of time and in turn misses its deadline. The priority ceiling protocol has been proved effective in real-time systems [101]. Careful extensions of current techniques can hence avoid re-inventing the wheel.

From a real-time systems perspective, providing hard guarantees for AI algorithms is difficult because of the large variance in execution time [126]. Such guarantees, if granted, may lead to very poor system utilization at run time. On the other hand, forcing developers to employ AI methods which have more predictable execution time will result in limiting the choice of (e.g. deliberative) algorithms and thus affect the usefulness of the application.

When AI systems have to operate under real-time constraints where approximation is necessary, one solution is to employ anytime algorithms mentioned above [44], which are a kind of imprecise computation (iterative refinement) [37, 83] for providing quality-time trade-off and approximate solutions. This is especially true when deliberations, such as reasoning and planning, are required since they can be highly expensive computationally.

However, the supporting task model can be a source of difficulty when *hard* guarantees are required:

consider an anytime planning algorithm [44, 69], the initial part of which requires a budget for guaranteeing a minimum acceptable plan to be constructed. The algorithm then refines the quality of the results whenever there are available resources. At some point it has to stop to meet its timing requirement and the final result may need to be written back to a shared memory area, an I/O device or sent over a network (e.g., in a cooperative multi-agent system). It then waits for the corresponding acknowledgement, which under real-time constraints may require a particular real-time access protocol for guaranteeing schedulability. Even using the "Mandatory-Optional" task structure in the original imprecise computation model [83], the final part of the computation is not guaranteed and may not finish before its deadline. Such computation can be regarded as the *action* component of the "Percept-Deliberation-Action" cycle in common AI agents [95].

The "Prologue-Optional-Epilogue" (P-O-E) task model proposed by Audsley el al. [11, 7], which is a generalization of that of Liu et al., can accommodate tasks of such structure. The authors devised schedulability tests, based on response time analysis with task offsets [8, 117], for guaranteeing schedulability of the mandatory (prologue and epilogue) tasks, where the optional components can be *unbounded*. However, their work mainly concerns system schedulability rather than maximizing the available time for scheduling optional components, which is constrained by the interval between the prologue and epilogue tasks.

With regard to this problem, one approach is to adopt a Slack Stealer [72] or a bandwidth-preserving server such as the Deferrable Server [113] for delaying the execution of hard tasks in favor of scheduling optional tasks. Unfortunately the Slack Stealer is characterized by very high overheads whereas bandwidth-preserving algorithms are not designed for this purpose and will introduce pessimism in the schedulability analysis. Dual Priority Scheduling [43] has been shown to be effective in scheduling aperiodic tasks with low overheads. In particular, the execution of hard tasks is deferred as much as possible based on the their worst-case response times. It is shown that the approach can also be applied to support imprecise computation by devising a modified promotion strategy for different types of tasks.

System support can also be an issue: most of the earlier scheduling algorithms and results in imprecise computation are EDF-based while the majority of existing real-time operating systems (RTOS) use static priority scheduling [27, 81, 109]. They therefore do not provide the necessary support for software designers to implement imprecise algorithms. Yet fixed priority response time analysis [10, 8], which this work is based upon, has been extended to accommodate various timing requirements such as task release jitter, offsets, arbitrary deadlines, etc., emerging as a mature real-time scheduling technology [23, 27].

This thesis shows how real-time AI applications can be supported by theories and tools available in fixed priority preemptive scheduling (FPPS).

1.2 A Case Study From RoboCup

Here a general problem of maximizing resource usage that many real-time AI applications have in common is presented. In particular, it is demonstrated through a case study based on a simplified version of the RoboCup simulation league [30] in which two teams of autonomous agents compete in a simulated soccer game. The RoboCup is chosen because it is very well known in the AI community for employing AI techniques in environments with real-time constraints, where many advanced and approximation techniques have been developed. Irrelevant details are ignored and only a simplified version with the necessary timing requirements that captures the essence of the problem is presented.

In the competition, there are two machines each hosting 11 instances of an agent program which connect to a referee server for soccer game simulation via a network connection. The agents in each machine have to decide an action command to be sent to the simulation server based on the environmental information they receive from the server. The programs can make use of any reasoning method; they just have to obey the protocol of the game. The simulation and perception of world state updates every 100ms and each agent is required to indicate a movement action every 10ms. Here the latter timing is taken as hard requirement which the agent program has to meet.

The problem is that, given a target machine for the agent programs to be run upon, how does one build the agent (to be run with 11 instances using the same program) such that the system resource usage can be maximized? In particular, each agent has to make an action decision every 10ms; how can one make sure all the "Percept-Deliberation-Action" computation of the 11 agents will finish within the timing constraint while not wasting any free system capacity?

If the AI methods employed are sophisticated, where the worst-case execution time is large, then the system may not be able to guarantee the schedulability of all agents. And if it can be guaranteed, since the worst-case rarely happens one runs the risk of serious under-utilization at run-time. However, using restricted methods also results in inefficient use of resources, decreasing the utility that can be gained by giving more time for agents to perform deliberative reasoning and planning.

It is for this reason that the machine provided by the official organization in the competition is very fast - fast enough to encourage more sophisticated methods to be developed by the AI researchers. Having said that, there will always be available resources that can be harnessed for enhancing agent's utility. How should these resources be managed?

1.3 Thesis Statement and Structure

The central hypothesis of this thesis is that significant improvements over current approaches in supporting real-time AI applications, based on theories in fixed priority preemptive scheduling (FPPS), are possible.

The remainder of this thesis is organized as follows: in the next chapter the related works in the area of RTAI, including various developed techniques and architectures for providing timely intelligent behaviors, are reviewed.

In Chapter 3 a review is provided for fundamental results in real-time scheduling, with an emphasis in response time analysis, and scheduling results in imprecise computation applicable for real-time AI applications.

Using a case study based on a scenario in the Robocup competition, the task model for imprecise computation that support the real-time AI systems concerned is presented in Chapter 4 where an exact schedulability test is derived. This result is based on Audsley's method with a simple offset value between the prologue and epilogue tasks, within the fixed-priority scheduling framework.

In Chapter 5, an enhanced scheduling strategy based on dual-priority scheduling is proposed. First the theory is presented where the applicability and performance are demonstrated. Potential ways by which aperiodic tasks can be scheduled are also discussed.

The exact schedulability derived is NP-hard in the worst case. For practical systems with many tasks a more efficient method is needed. A tractable and sufficient, but not exact, schedulability test is formulated for solving this problem in Chapter 6 where performance evaluation is provided.

Determining task priority easily is important in flexible real-time AI systems. In Chapter 7, an optimal priority ordering algorithm is presented where it is shown that given a task set ordered by task utility which is feasible, the algorithm will find it with the property that the lexicographical distance to the optimal ordering will be minimized.

Finally the thesis is concluded and future research directions are discussed.

Chapter 2

Literature Review: Real-Time AI

Real-Time AI (RTAI) is a sub-area of AI research concerning about real-time performance of AI systems. In general, AI research can be thought of as formalizing and modelling human intelligent behaviors including the abilities to perform complex reasonings. Solving problems by search is a common approach in AI, which can prove time-intractable with just moderate problem size. One major problem, in terms of real-time scheduling, is that they have effective unbounded worst-case execution time (WCET).

Although resource constraints were not an issue in traditional AI research, it becomes very important in real-world domains such as medical care, robotics and military operations where there are stringent time requirements that a system has to satisfy. When they have to operate in some dynamic environment the time available for computation varies from time to time, and from situation to situation.

Consequently, various complexity-limiting techniques have been developed in the AI community. These techniques include: 1) satisficing - Anytime Algorithm and, Multiple and Approximate Methods; and 2) Meta-reasoning. In addition to particular RTAI techniques, another approach is to build systems consisting both real-time and AI components. In this chapter these techniques and attempted RTAI architectures are reviewed.

2.1 Anytime Algorithms

The term Anytime Algorithm was coined by Dean and Boddy [44]. Anytime algorithms refer to the class of algorithms that the quality of computation, in terms of completeness, precision or certainty, improves gradually over time. In particular, they are useful in real-time situations because they always have an answer at hand. Many problems in AI, such as planning, are intractable such that optimal results can not be obtained within reasonable amount of time. However, although getting an optimal solution is intractable, a good partial result may be obtained in a much shorter time. In essence, anytime algorithms provide a mechanism of “satisficing”. In addition, having the ability to reason about how much time is needed for getting a solution of acceptable quality, can make an agent more adaptive to a dynamic environment.

Due to the relatively general definition of anytime algorithm many other algorithms with similar property also fall into this category. For example, one can argue that a simple breadth-first search algorithm is a kind of anytime algorithm if the best-so-far solution is always kept and returned when needed. Other algorithms with anytime characteristic include numerical approximation, dynamic programming, and Monte Carlo algorithms.

Because the time in searching a particular solution can be different each time it is performed, time-variations can be unpredictable [114]. A real-time heuristic search technique was proposed by Korf [69], called Real-Time A* (RTA*). RTA* introduces a time constraint associated with node expansion which determines the lookahead time allowed in making a decision. The algorithm is proved to make a locally optimal decision and is guaranteed to find a solution with certain assumptions. There are other techniques in depth-constrained searching that may be useful in real-time applications; examples are progressive reasoning [121] and depth-first iterative deepening [68].

One major problem in using anytime algorithm is in generating *performance profiles* of search-based algorithms which allow backtracking [53]. A performance profile is a function that returns the expected quality of an anytime algorithm when given a certain amount of time. In non-search domains with well-defined error functions, precise performance profiles may be obtained. This is similar to the Imprecise Computation without the mandatory part; imprecise computation is discussed with more details in Section 3.4.

2.1.1 Deliberation Scheduling

Dean and Boddy [44] considered the problem of time-dependent planning in which the time available for planning varies from situation to situation. Deliberation scheduling is a procedure used to explicity allocate resources to deliberation tasks, which are anytime algorithms, in order to maximize the total utility of actions chosen in deliberation. With the assumptions they devised, they gave two different optimal algorithms in scheduling anytime tasks. Because tasks are assumed to be *independent*, the scheduling algorithms are computationally inexpensive and are therefore not included as part of the cost for scheduling.

2.1.2 Composition and Compilation of Anytime Algorithms

A real practical system is often consisted of many layers and subsystems with many dependencies. Zilberstein and Russell have been formalizing many aspects of anytime algorithms including composing system in which the output of a set of anytime algorithms can be the input of another anytime algorithm, whose quality profile is represented as a conditional performance profile [97, 124]. Given a specific amount of time, a compilation of anytime algorithms is the best allocation of time among individual algorithms so that the expected utility is maximized. Such a compiled program is called a contract anytime algorithm, which states that the amount of time for execution must be told in advance. Otherwise no answer of expected quality is guaranteed to be returned.

It was also shown that if unrestricted dependencies are allowed between performance profiles, the complexity of compilation is NP-Complete in the strong sense [124]. However, if the dependencies are restricted to linear or tree composition, there are local heuristic (in pseudo-polynomial time) compilation algorithms yielding globally optimal results, assuming that all conditional performance profiles are *monotonic non-decreasing* function of their input. In addition, they showed that any usual anytime algorithm can be constructed from contract anytime algorithm with a penalty of execution time at most four times greater [97].

An immediate issue one may see is the time complexity of compilation may be prohibitive for online use. As in the time-dependent planning problem, the time that is available for execution may vary depending on different situations. Offline compilation seems to be inflexible since in real-time systems, there may be available slack from the system dynamically due to hard tasks not using all the WCET and sporadic tasks not arriving at the maximum rate. Fixed compilation will not be adaptive enough to make use of dynamically available resources or make tradeoffs when anticipated resources are not available because of the nature of contract anytime algorithm. Moreover, the assumption that partial result from an anytime algorithm can be used as input for another anytime algorithm may not be justified in practical applications.

2.2 Multiple Methods and Approximate Processing

In multiple methods, a set of methods is used to make tradeoffs between quality and time instead of a single anytime algorithm which improves gradually over time. The methods may have different performance characteristics depending on the environment. This tradeoff mechanism was also termed *approximate processing* by Lesser [75].

Multiple methods have two important advantages over anytime algorithms. The first one is that sometimes it may be difficult to find an algorithm that improves predictably and monotonically over time, this is however not an issue for multiple methods. Another one is that they do not have to provide quality/time tradeoffs only - different methods for a task can be entirely different depending on the problem's nature. Note that the imprecise computation with 0/1 constraint and the exception-handling in real-time system research can be seen as special cases of multiple methods with only two versions [81]. However, a potential advantage of anytime algorithms over multiple methods is that usually one anytime algorithm of each type of task has to be programmed, whereas in multiple methods there may be several versions for each task. This may reduce the programming time needed and the likelihood of errors.

The Design-to-Time [52] approach assumes the existence of multiple methods for tasks and considers the problem of designing a solution (a schedule), from the available methods, that uses all available resources to maximize solution quality. In the task model, there are groups of tasks in the system and tasks can further have subtasks recursively. Tasks are allowed to have interdependencies similar to that considered by Zilberstein's compilation of anytime algorithms [97, 124]. The name comes from the fact that it tries to use all available time to generate the best solutions possible. The approach relies heavily on accurate monitoring of tasks' execution and intermediate results sharing among tasks. That is, system support becomes very important. In the case of small set of tasks, near linear time performance for designing solutions can be obtained. However, when task set size and interdependencies increase, the algorithm does not scale well. One way to work around is to use tradeoffs made between the schedule precision and the schedule-deliberation time, which exhibits the contract anytime algorithm character.

The multiple methods provide more discrete tradeoffs rather than the linear-like property of anytime algorithms. This may or may not be a desired property although it can be seen that less research has been done on multiple methods compared to anytime algorithms. Nevertheless, multiple methods seem to be a more general approach for that incremental improvements in anytime algorithms can be thought of as many individual methods. Undoubtedly, more extensions are needed to the current imprecise computation research for representing and scheduling the more general multiple methods developed in RTAI.

2.3 Decision-Theoretic Meta-reasoning

Decision-theoretic meta-reasoning [94] applies the theory of information value to select computation. The value of computation depends on both its costs and benefits. For example, one can guarantee a searching algorithm to have anytime characteristics automatically if one designs it with a meta-reasoning technique. However, meta-reasoning can be expensive computationally where heuristics, compilations of reasoning strategies and decisions may need to be implemented.

Horvitz considered the general problem of partitioning resources between meta-reasoning and problem solving [61]. The problem-solving itself involves planning and base-level computation. He found optimal mathematical solutions to various partition problems by assuming the characteristic functions to be in some particular forms. However, he ignored the *meta*-metareasoning cost (analogous to the negligible scheduling costs in real-time scheduling) because the complexity is constant in time - solutions are only computed by plugging values into simple equations with the defined assumptions.

Since satisficing techniques seem to be a more natural approach than meta-reasoning, research has been focused on anytime algorithms and multiple approximate methods. One reason may be that anytime algorithms have the advantage that they can be stopped anytime in an emergency; while meta-reasoning seems to require more information *a priori*, which may not be flexible enough in a dynamic environment.

2.4 RTAI Architectures

The above techniques can be seen as trying to be as "intelligent" as possible in the available time. Another way of being both deliberative and reactive at the same time is to build a system consisting of asynchronous subsystems - reactive real-time subsystem for fast responses and deliberative subsystem for intelligent adaptive behaviors [53], examples are PRS [65], Phoenix [62] and Guardian [57]. This architecture, as argued in [88], seems able to solve the weaknesses mentioned above because it imposes least limitations on the techniques from the two areas.

However, most so called "real-time" AI systems, including the Soft Real-Time Agent (SRTA) architecture [119] for instance, do not run on a real-time operating system (RTOS) but only on a genereal purpose operating system [59]. The authors noted that the agent is only statistically "fast enough" and although individual components may be able to run efficiently but when they are opearating together, the time variance can be so large that important tasks may miss their deadlines.

This is clearly not desirable in safety-critical domains such as medical care and military applications where AI techniques are being applied. Therefore, it becomes increasingly necessary to provide a suitable hard real-time, yet flexible infrastructure for implementing real-time agents - before the general purpose operating system becomes the norm.

CIRCA (Cooperative Intelligent Real-Time Control Architecture) [87] is also a two-level system that can provide hard real-time guarantees by only making plan that is executable by the lower system in time. More specifically, in the reasoning process, the AI subsystem reasons about actions with their known execution times and produces a plan consisting of individual actions in such a way that the plan can be implemented by the real-time subsystem by some deadline, and the two subsystems cooperate through a communication interface. Nevertheless, the real-time subsystem in CIRCA is a simplistic one that only runs a cyclic executive plan supplied by the AI subsystem. The disadvantages of using a cyclic executive are well known in the real-time systems community [12].

2.5 Practical Issues

As noted above, a common problem among current real-time AI systems is that they only run on general purpose operating systems (GPOS) [88, 60], which is only soft real-time at best. Hard guarantees, which are of major concern in real-time systems, are not provided. For example, contract anytime algorithms [124] require strict resource reservations a priori otherwise the quality of the results produced is not guaranteed; this guarantee however can not be provided by a GPOS. Zilberstein [126] also considered a resource allocation profile for a composite system consisting of a set of anytime algorithms. However, it is only valid when there are no other processes in the system since only the resource allocation among the anytime algorithms is considered. New processes can enter into the system at any time and common resources may also be held without a well-defined upper bound duration, effectively breaking down any optimality in such predefined allocations.

Satisficing techniques are still not a mature and popular technology in practice. Little research efforts have been witnessed from the real-time community since the works in early 90's discussed above. In the AI community, recent research has been focusing on making programming anytime algorithms easier [54, 56], combining different satisficing methods together in a system, and building RTAI component to be embedded in larger real-time systems [53]. RTAI research has been focusing on defining and elucidating particular satisficing techniques such as anytime algorithms and multiple methods. These techniques, when used alone, have very limited practical advantage. Thus a more hybrid and heterogenous approach is necessary, for example, solving a particular problem where the combination of both anytime algorithm and multiple methods is better than using each technique alone.

Currently, satisficing techniques tend to be *ad hoc* and only customized to the particular problems for which they are designed to solve [53]. Hence, they are generally not reusable or portable to other AI systems. Problems also arise when they are integrated into larger system in which they have to cooperate with other components in the system, where the assumptions and so-derived optimality they held may be no longer valid. The compilation of anytime algorithms may have to be performed at runtime in some dynamic environment in which the time available for execution can not possibly be known a priori. More efficient compilation method needs to be devised or, if still inefficient, heuristics may need to be employed.

One real-time AI system that provides better system-level guarantee is the CIRCA platform as described earlier. However, the disadvantages of using a cyclic executive at the scheduling level, such as inflexibility in programming, are well known in the real-time systems literature. The approach employed in this thesis is more integrated where tasks can be facilitated directly by more flexible scheduling in the operating system, which is supported with schedulability theory in fixed priority scheduling.

2.6 Summary

As mentioned before, the more advanced scheduling technologies in real-time systems can be useful in supporting real-time AI applications. At the moment AI researchers tend to ignore these possible options and focus on restricting AI techniques to be real-time for high level problem-solving [59]. For example, data sharing and resource accessing control have always been issues in real-time systems and effective solutions have been developed. RTAI applications may be facing similar problems in the near future and therefore more real-time research results should be considered.

In real-world safety-critical applications, periodic control and monitoring processes are often necessary. These can be safely scheduled and guaranteed by traditional real-time systems. When cognitive and intelligent behaviors are needed, an AI subsystem can be built upon the real-time system. In such architecture cooperations between two systems are needed. For example, slack time is sometimes available in the base real-time systems due to tasks not using up the guaranteed time resources. Satisficing algorithms can possibly use this slack for improving result quality.

Run time monitoring is essential in using anytime algorithms and multiple methods because one must stop running an algorithm when time is up or measure whether the output quality of the algorithm is the same as expected [124, 88]. Many of these functions run at a lower level that they can be delegated to the base real-time systems because more accurate control can be achieved due to the finer granularity at the system level. Such issues have been addressed by the real-time systems community in the past where system-level supports have been well studied.

In the next chapter, the relevant scheduling theories and results, mainly based on fixed priority preemptive scheduling, are reviewed with an emphasis on supporting real-time AI systems.

Chapter 3

Literature Review: Real-Time Systems Scheduling

In the first chapter the need and advantage of using real-time technologies to support deliberative agents were briefly mentioned. In this chapter real-time scheduling theories and recent developments are reviewed along with their assumptions, validity and limitations. Since scheduling is the fundamental means of providing resource allocation and management in a real-time system, to support deliberative agents that make tradeoffs depending on available resources, one has to look at scheduling.

In general, the followings are the minimum requirements that a real-time system should meet [110, 27]:

- all tasks in the system be schedulable using average execution times and average arrival rates;
- all hard tasks are schedulable using worst-case execution times and worst-case arrival rates of all tasks in the system.

A deliberative agent will require the system, in addition to the above objectives, to [26]:

- maximize system utility by making use of all available resources - including spare capacity and dynamically available resources such as gain time[1];
- minimize tradeoffs whenever they are inevitable.

[1]And, at the same time, recognize the *limitation* imposed by the capability of the system.

It is assumed that an agent is implemented as an ordinary application in a system, running upon an operating system in which other applications and system-level tasks exist that are *not concerned* by the agent, but are necessary for proper system functioning. Within such context, with regard to bounded optimality, the agent tries to maximize its utility by making use all the resources enabled by the hardware architecture (processor speed, etc.) and all the remaining resources from the system after guaranteeing all hard tasks meet their deadlines.

Like a traditional real-time application, a RTAI application also needs to deal with issues such as aperiodic task scheduling, resource access control, task interdependencies, resource reclaiming, etc.; unlike a usual real-time application, it has to maximize its overall utility by making use of all available resources, and make tradeoffs depends on particular environment, which is not a conventional issue that real-time systems address. Hence in the real-time systems literature, there already exist useful technologies with, nevertheless, different limitations for facilitating satisficing. The related works are surveyed in the literature broadly according to the following categories:

- Guaranteeing hard periodic tasks;
- Imprecise computation;
- Scheduling soft aperiodic tasks with spare capacity;
- Multiple applications and hierarchical scheduling;
- Practical issues: programming language and system support.

In particular, fixed-priority scheduling theory is, among others, the focus of this review, due to its matureness and existing system supports [8, 6]. The works to be mentioned in different categories below are all closely related to facilitating satisficing techniques. We describe their relations to, and when appropriate, their prospect in supporting RTAI applications. In addition, the consideration is restricted to single processor systems.

3.1 Background

There are broadly two types of timing constraints in real-time systems: hard and soft. A task is hard if it is critical and requires absolute guarantee that the system will meet all its timing constraints; a soft task is one that whose timing constraints, when missed, does not cause serious negative effects.

A schedule is feasible if all tasks in it completes by their deadlines. A set of tasks are schedulable if there is at least one algorithm that can produce a feasible schedule and “schedulable by a particular scheduling algorithm” if the algorithm always produces a feasible schedule when used. A scheduling algorithms is optimal in the sense that it always produces feasible schedule whenever one exists. Feasibility is the dominant metric in hard real-time systems. One can also use other kind of metrics such as average and maximum tardiness, average response time for soft tasks.

A schedulability test is a validation method for ensuring a set of tasks is schedulable. Offline schedulability analysis ensures a system will meet all its timing constraints and it is always performed for validation in hard real-time systems before the system actually runs. In open systems where new requests can arrive, online acceptance test can be performed before admitting them into the system.

A test is called *sufficient* if the set of tasks which passes the test is indeed schedulable; a test is called *necessary* if a set of tasks failing the test is in fact unschedulable. An *exact* schedulability test is both sufficient and necessary.

Offline scheduling refers to generating pre-computed schedules before the system actually executes. At run time, the system simply follows the schedule of what task to execute. The scheduling is done offline. An example is a Cyclic Executive. Offline scheduling requires information about all the tasks be known a priori and is inflexible if the future workload of the system is unknown. Many other problems with Cyclic Executive scheduling are documented in [84]. Nevertheless it is suitable for environment which is completely deterministic; most safety-critical systems used cyclic executive until only recently [12].

Online scheduling algorithms make each scheduling decision only when the parameters of a task becomes available after it is released. One example is the fixed-priority scheduling described in this section. As mentioned, this kind of algorithm is crucial in dynamic environment that we concern. Note that scheduling results in this section only apply to single processor scheduling unless otherwise stated.

3.2 Computational Model and Notations

The well known periodic task model [80] is a simple model without aperiodic and sporadic tasks. The followings are the assumptions used:

- there are only hard periodic tasks, with known period;

- tasks are independent of each other;
- all tasks have deadlines equal to their periods in each invocation;
- all tasks have fixed worst-case execution time;
- tasks are preemptable;
- tasks can not suspend theirselves during execution;
- context switch times are ignored.

There are n number of tasks in the system. A task τ_i has distinct priority i, so that $1 \leq i \leq n$ (the smallest the number the highest the priority), and is basically characterised by:

- Period T_i: the minimum arrival interval of task;
- Computation time C_i: the amount of time the processor needs to execute the task without interruption;
- Deadline D_i: the time before which task should finish its execution.

$hp(i)$ is used to denote the set of tasks with higher priority than task τ_i. In addition, when applicable:

- Utilization U_i: the utilization of a task, which equals C/T;
- Interference I_i: the time interference caused by preemption of higher priority task;
- Blocking time B_i: the worst case blocking time due to waiting on lock release of semaphore;
- Release jitter J_i: the bounded delay of actual release of a task;
- Response time R_i: the worst-case finishing time after the release of a task;

3.3 Guaranteeing Hard Periodic Tasks

Guaranteeing hard periodic tasks is the first and foremost requirement in hard real-time systems; and it will continue to be imperative in future safety-critical systems. In general, finding optimal scheduling policy (the priority assignment algorithm and corresponding schedulability test) is a primitive objective in real-time scheduling for verifying a set of hard tasks will meet all their timing constraints.

3.3.1 The Liu and Layland Analysis

With the simple periodic model, in 1973, Liu and Layland [80] showed one can analyze the schedulability of a system by considering only the utilization of a task set. The total utilization U of a system is defined to be:

$$U = \sum_{\forall i} \frac{C_i}{T_i} \tag{3.1}$$

They show that Rate Monotonic (RM) priority ordering, which assigns priority according to task period (the smaller the period the higher the priority), is optimal in any static priority assignment when the simple model is assumed and that deadline is equal to period ($D_i = T_i$).

A sufficient schedulability test was given [80] when tasks are ordered according to the rate monotonic policy:

$$\sum_{\forall i} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{3.2}$$

As the number of tasks increases, the bound tends to $ln(2)$, which is about 69.3%. This is a simple and powerful test because we know that as long as the total utilization is below the 69.3% bound, the system is schedulable and all tasks will meet their deadlines; but failing the test does not mean the task set is not schedulable.

The rather restrictive computation model limits the applicability of the result. For example, it is reasonable for tasks to have their deadlines before their period. Leung and Whitehead [76] showed that rate monotonic policy is not optimal in this case, but the Deadline Monotonic (DM) priority ordering by which task's priority is assigned by the highest priority to the smallest relative deadline. If assumptions of the task model are to be relaxed the simple schedulability test above can not be applied. It is obvious that for advanced AI systems, in which tasks may be nonpreemptable, have precedence constraints and share common resources, the simple task model can not capture the actual scenarios and thus can not be used. Later research in real-time systems extended the Liu and Layland model by relaxing various assumptions imposed on the model.

3.3.2 The Exact Response Time Analysis

The simple test above is sufficient but not necessary; that is, it is pessimistic and it was shown that on average the utilization bound for a large set of tasks, based on RM, is about 88% [74], compared to the worst-case 69.3%.

A more accurate schedulability test can enable a system to schedule more different task sets. Rather than using utilization, there is an alternate way to determine the schedulability of a system. The response time analysis [10, 8] depends on calculating task's worst-case response time by taking into account all the possible interference that can happen due to higher priority tasks being released during a task execution. One way to calculate the interference I_i that a task can suffer is to compute the time used by all higher-priority (hp) tasks released before its deadline D_i:

$$I_i = \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \tag{3.3}$$

This account of interference is sufficient, but, not necessary. Initiated by Harter [89], then later by Joseph and Pandya [66], and Audsley et al. [10, 8], an exact method is used for evaluating interference. The longest response time R_i of a task τ_i is determined, at the critical instant[2], as:

$$R_i = C_i + I_i, \tag{3.4}$$

where

$$I_i = \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.5}$$

Therefore,

$$R_i = C_i + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.6}$$

So a task τ_i will meet its deadline if:

$$R_i \leq D_i \tag{3.7}$$

And a system is schedulable if and only if for all tasks the above relation holds.

However, solving equation 3.6 is not trivial as the term R_i exists at both sides of the equation and the term R_i in the right side is inside a ceiling function. It is a fixed-point equation for which many solutions may exist; but the smallest (non-negative) solution corresponds to the worst-case response time for a task. Audsley et al. [8] proposed an efficient method for solving equation 3.6 by forming a recurrence relationship:

$$R_i^{n+1} = C_i + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{3.8}$$

At each iteration the value R_i^{n+1} is re-evaluated using the previous R_i^n and increased in a monotonic non-decreasing fashion. When $R_i^{n+1} = R_i^n$, the solution is found. If the found value is greater than R_i^0

[2] A time at which all higher priority tasks also released at the same time (the worst-case)[80].

(which can be set to 0 or C_i initially) then it is the value we want. If there is no solution found then R_i will continue to increase - this implies it is a low-priority task which continue suffering interferences from higher priority tasks and can not be scheduled in the system.

The response time analysis is pseudo-polynomial in complexity but is superior to the utilization-based test in the sense that it is both sufficient and necessary. It allows tasks having deadlines less than period as well as arbitrary priority assignment policy because it does not make any assumption on the task's priority assignment. It is flexible and has been extended to account for different application requirements, such as resource access blocking time [101], release jitters and arbitrary deadlines of tasks [118], as well as fault tolerance (by trivially adding an extra term in the equation), which make it a very versatile real-time scheduling technology [6].

3.3.3 Allowing Task Interdependencies

Besides the relaxations on task period and priority assignment, another relaxation of the task model is task's interdependencies. When tasks are allowed to access or synchronize mutual exclusive resources, blocking can happen due to the mechanism of resource locking. In hard real-time systems any blocking time must be bounded for accurate analysis. Although it can be solved by making critical section non-preemptable [85], one particular problem in fixed-priority systems is that of *Priority Inversion* where a high priority can be blocked (for an arbitrary amount of time) by a medium priority task because a low priority task has locked a resource needed by the high priority task [71, 100]. This is clearly not a desired property in a priority-based system because higher priority tasks are supposed to be more critical.

Priority Inheritence Protocol [100, 101] (PIP) was introduced by Sha et al. for solving the priority inversion problem. In particular, the protocol specifies that if a higher priority task is blocked by a low priority one, the block-causing task should execute at the maximum possible priority - either of its own or the highest priority of the tasks that it is blocking. The approach, however, does not prevent deadlock and multiple blocking. These problems are solved by introducing an additional rule to PIP: to associate a priority ceiling to each shared resource, resulting in the Priority Ceiling Protocol (PCP) [100, 101].

Desirable features of PCP as a resource control protocol includes:

- A high-priority task can be blocked at most once during its execution by lower-priority tasks;
- Deadlocks are prevented;

- Transitive blocking is prevented;
- Mutual exclusive access is enforced by the protocol itself.

PCP states that each resource is associated with a priority ceiling, which is equal to the highest priority of all the tasks that locks the resource. A task has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority tasks. A task can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (not counting any resources it has already locked).

Because the worst-case blocking time can be determined in PCP, one can incorporate into the response time analysis by introducing a blocking term B_i [10]. The response time equation is thus:

$$R_i = C_i + B_i + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.9}$$

where

$$B_i = \max_{k \in K}\ usage(k,i)C(k) \tag{3.10}$$

K is the total number of critical section and $C(k)$ is the worst-case execution time of critical section k. The function *usage* returns either 0 or 1 - 1 if there is at least one lower priority task and at least one equal or higher priority task that uses the resource k.

Equation 3.9 can be solved by forming a recurrence relation mentioned above. However it may now be pessimistic with the blocking term because whether a task will suffer the maximum blocking depends on the actual phasing of task. That is, the test may not be both sufficient and necessary. However, generally the equation provides an effective means for performing scheduling analysis [27].

Precedence constraint between tasks is another kind of task interdependency. Two tasks are said to have precedence constraint if the successor can not execute before the predecessor has finished its execution. There are not many scheduling results on tasks with precedence constraints. Audsley et al. noted that precedence constraints can be enforced by suitable choice of task offset and period implicitly [5]. They have also considered the priority assignment and grouping of precedence-constrained tasks where tasks are in linear composition [9, 12].

However, anytime algorithms may have differrent kinds of precedence constraints. In particular, they can be hierarchical [97, 124]. The clustering of tasks with local functions and precedence constraints mentioned is a powerful concept. For example, hierarchical composition can be thought of as *nested* clustering from the system scheduler's point of view. If local analysis can be performed without considering all other tasks in the system, the global analysis problem may be eased significantly (e.g. from intractable to tractable). Groups of tasks may be better supported by different *periodic servers*, which will be further discussed in the multiple servers section (Section 3.5.3).

3.4 Providing Flexibility: Imprecise Computation

There is an intimate relation between the satisficing techniques in RTAI and imprecise computation in real-time scheduling. Imprecise computation is closely related because it also provides tradeoffs between quality against time analogous to anytime algorithms.

Conventional real-time system requirements focus on ensuring that a set of tasks will satisfy all its deadlines, be they periodic or sporadic tasks. Scheduling analysis is done offline, at design-time, in a static way that guarantees schedulability of systems in runtime. In practise, however, some unpredictable, dynamic situations may occur such as transient overload, tasks overrunning their WCET due to I/O dynamics, and unavailability of anticipated resources. Imprecise computation is an approximating approach that can deal with these runtime dynamics where time and resources are not enough for computation before certain deadlines. It provides a well-defined mechanism with which partial results can be returned when needed, producing a smoother transition as an error handling technique, without which dynamic failures may lead to catastrophic consequences.

Imprecise computation in real-time systems has been studied by Liu et al. extensively [37, 83, 104, 103]. Computation which allows approximate results has been used in computer systems for many years; for example numerical approximation. However, there has been no formal notion for performing tradeoff before. In many situations, one is willing to accept approximate results with inferior quality when the results of desired quality can not be obtained due to timing constraints or unexpected resource shortage. This is particularly true in hard real-time systems where unpredictable situations may arise at runtime no matter how much amount of offline analaysis has been performed.

In the imprecise computation model proposed by Liu et al. [83, 104], a (periodic) task τ_i is composed of infinite invocations of *jobs* e_i which in turn consists two subparts - a mandatory part $e_{m,i}$ and an optional part $e_{o,i}$. Figure 3.1 illustrates the task structure of which mandatory part is executed before the optional part (precedence constraint).

All mandatory subtasks are critical and have to be guaranteed in the system; whereas optional subtasks,

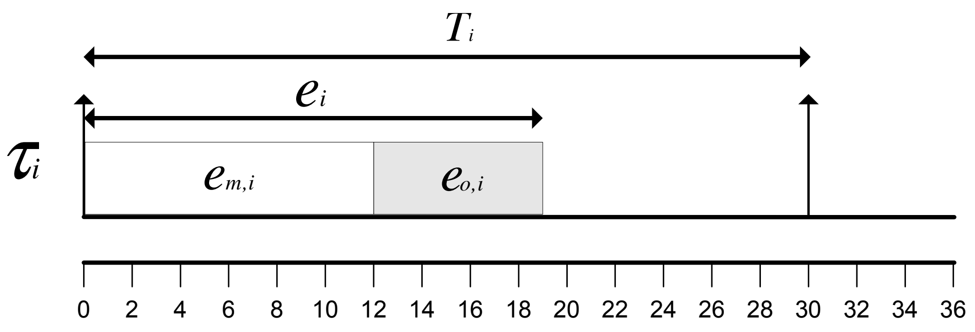

Figure 3.1: Structure of an imprecise task.

which are used to refine the results computed in the mandatory part, are skippable. Algorithms are designed to be monotone (a fair assumption) - that is, the quality of the computation produced by an algorithm is improved in a non-decreasing manner as more time is given. Monotone algorithms are more flexible and suitable for analysis. The amount of optional subtask not executed is defined to be the *error* ϵ of a task.

Error functions can be categorized into three different types, namely linear, convex and concave. Figure 3.2 illustrates the different "shapes" of error functions. When the optional subtask of a task is executed to completion, the error is said to be zero. Linear error function is defined as:

$$\epsilon(x) = 1 - \frac{x}{e_{o,i}} \tag{3.11}$$

where x is the amount of time executed of the optional component.

When all the parameters of all the tasks in a system are known, optimal schedule can be computed offline at design time. The following results can be used to find schedules for particular optimizations offline. By using the EDF algorithm, according to rate-monotone policy [80], all the mandatory subtasks $e_{m,i}$ of a task τ_i are guaranteed schedulable if [37]:

$$\sum_{\forall i} \frac{e_{m,i}}{T_i} \leq 1 \tag{3.12}$$

Of course, when all the mandatory subtasks and optional subtasks of all tasks in the system consisting of solely periodic tasks, when total utilization is below 1 and scheduled by EDF, all tasks will meet their deadlines and the error will be zero assuming deadline is equal to period ($D_i = T_i$).

If an optional subtask is only scheduled in background and whenever mandatory subtask is ready it is executed first, the schedulability of any mandatory subtasks will not be compromised. The objective is

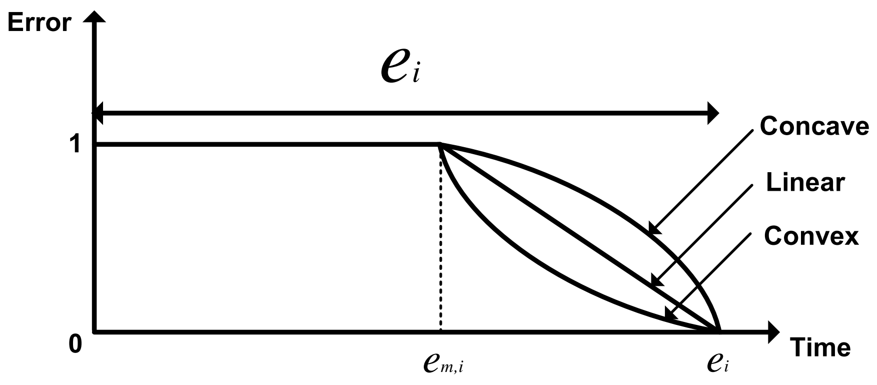

Figure 3.2: Types of error functions.

then to allocate optional subtasks in the schedule to optimize certain defined metrics, such as total error, average error and maximum error. The following concerns complexity in constructing optimal schedule offline.

Assume error functions are linear, in the simple case that the total (identical weight) error is to be minimized, Shih et al. have shown the complexity of finding the optimal schedule is $O(n\ log\ n)$ for single processors [104] and $O(n^2 log^2 n)$ in general [102]. When the error functions are convex and weights are identical, they showed that an optimal schedule can be found in $O(n^2)$ time (optional subtasks with identical execution time) and $O(n^3)$ (optional subtasks having arbitrary execution times).

When tasks are allowed to have arbitrary weights, which is more realistic in real world applications, the problem is more complicated. If minimizing total weighted error is considered, where the problem can be reduced to a minimum-cost-maximum-flow problem [102], the time complexity is $O(n^2 log^3 n)$ on multiprocessors. Shih et al. [104] later showed a faster algorithm, in $O(n^2 log\ n)$, for single processors. Leung et al. have given the fastest algorithm which runs in $O(n\ log\ n + kn)$ time, where k is the number of different weights [78]. The problem of minimizing the maximum weighted error has been studied by Ho et al. [58]. They showed an algorithm with time complexity of $O(n^3 log^2 n)$ for multiprocessors and $O(n^2)$ for single processors.

When the error functions of some or all tasks are concave or subject to 0/1 constraints, finding optimal schedule with arbitrary weights is NP-hard in general. Liu [81] noted that performance data showed that in multiple methods, there is little advantage for anything beyond two versions. Further, two versions can be adequately represented by imprecise task with 0/1 constraint (i.e., either the optional subtask is run to completion or completely discarded), where the time for executing the optional part will be the time difference between the two versions. Unfortunately, Shih et al. [104] showed that the problem of scheduling such tasks, and minimizing the total processing time of the discarded optional subtask is NP-Complete even for a single processor.

Although Aydin et al. [14] showed that scheduling optional subtask in background to mandatory subtask is not optimal, however, using the background approach is more robust in the presence of overruns. In systems where mandatory subtasks are hard real-time it is more reliable - that mandatory parts are less likely to miss their deadlines rather than scheduling all tasks purely by EDF without discriminating different tasks.

Shih and Liu [103] proposed an algorithm for scheduling imprecise computation online which is similar to the slack stealing (Section 3.5) algorithm. The algorithm schedules both offline jobs and online jobs (whose task's parameters are only known at run time) with arbitrary release times and deadlines to minimize total error, in polynomial time, which is applicable only to certain restricted classes of problems. They also show that there is no optimal online scheduling algorithm for minimizing total error in the sense that it can always find a valid schedule when one exists.

Baruah and Hickey [16] studied the competitiveness online scheduling algorithms for imprecise computation. The study concerns overloaded systems with firm tasks where guaranteeing mandatory components is not a necessary requirement. This relaxation to the execution environment is used for devising a lower bound on the competitive ratio of scheduling algorithms. An algorithm was shown to behave with the complexity of the lower bound and they argued that the ratio serves as a good measure for online scheduling algorithms for overloaded system.

Leung et al. have studied other problems of imprecise computation scheduling, such as mean flow time and dual criteria optimization; more details can be found in [77].

The imprecise computation model assumes that when the optional component is executed up to completion, there will be no error produced. There is therefore a bound for execution time of the optional subtask. Audsley et al. [11] have investigated how to integrate "unbounded" software components into hard real-time systems, in particular, into the response time analysis [10, 8].

An extended task model is used to represent different types of computation. Rather than using the "Mandatory-Optional" structure, a "Prologue-Optional-Epilogue" model is employed. More specifically, a general computation is composed by five components:

$$I, C1, X, C2, O \tag{3.13}$$

where I and O stands for task input and output, $C1$ and $C2$ for mandatory computation, and X optional computation. Milestone method (imprecise computation), sieve function, or multiple versions (methods) can then be represented with different arrangements of the above parts.

A task τ_i is represented by 3 different components; the prologue (τ_i^p), epilogue (τ_i^e) and optional (τ_i^o) component where:

$$\tau_i^p = I, C1 \tag{3.14}$$

$$\tau_i^p = C1, O \tag{3.15}$$

$$\tau_i^o = X \tag{3.16}$$

Ignoring the schedulability of optional component (due to the nature of unbounded execution), the schedulability of mandatory computation (prologue and epilogue subtasks), even with shared resources which were not concerned before, can be determined by using the response time analysis mentioned in Section 3.3.2. Having guaranteed the hard components, different scheduling strategies can then be adopted to allocate resources for optional computation.

The response time test is sufficient but not necessary if τ_i^p and τ_i^e are considered as a single computation. Since the pair will not be executed at the same time a straight-forward application results in pessimism. A less pessimistic test has therefore been devised by using the offset analysis performed by Tindell [117], also in [7]. This model is discussed again in Chapter 4.

However, all the mentioned works above assume the availability of a single measure of utility across all tasks in a system without concerning how to obtain them. Error functions for describing the utility of optional computation are relatively simple in imprecise computation. In agent systems, where tasks are of different utilities and goals, it may not be easy to have a simple utility evaluation function for comparing all the tasks in a system.

Yen and Natarajan [122] tried to give a utility-based decision-theoretic formalism on imprecise computation in order to provide better justification for its usage in real-time systems. In particular, decision theory, which allows probability, is applied as an alternative for assigning utility for imprecise tasks. This work is much more applicable to RTAI applications because it is more commonly used in AI where task's utility can be unpredictable and can change constantly. In addition, the authors extended the imprecise task model to be a hierarchical one which can better accommodate AI tasks such as planning. However, they mainly addressed the utility problem without giving the corresponding scheduling algorithms.

Burns et al. [25] have proposed a framework for performing value-based scheduling, based on defining different behavior modes and using theories from measurement theory, and multiple-criteria decision making, for value assignment. Generally speaking, the problem of obtaining and defining utility function in real-time systems which allows tradeoffs remains to be explored. However, researchers are more aware of the importance of employing theories which are already established in other fields.

Besides providing more flexible scheduling, imprecise computation has also been applied in flexible resource allocation, QoS management, overload management [123], fault tolerance [13], real-time database query [120], real-time image and video transmission [63] etc.

As Audsley et al. [7] pointed out, the wide-spread use of imprecise computation will only happen if they are integrated into standard software engineering methods (see also Section 3.7). Besides, industrial experience is also important in justifying to what extent imprecise computation is indeed a proper technique in constructing flexible real-time systems. Nevertheless, formal mechanism that allows tradeoffs between computation time and quality does provide a natural way for employing complex AI algorithms in situations where intelligent behaviors are required.

Note that the distinction between anytime algorithms and imprecise computation is still blurred - some researchers argue each of them can be a generalisation of the other [114] whereas some argue that anytime algorithm can be represented by imprecise task consisting of solely optional part so that imprecise computation is a generalization of anytime algorithm [81] - perhaps what is needed is just a more unified formalism in the future.

An anytime algorithm is more like a soft task and the more time the system can provide the more it can use to improve result quality - the optional computation is viewed as *reward* rather than *error*. Hence such computation has no clear bound to which, when reached, the computation will produce perfect result.

Besides, the use of a periodic server has become more common in real-time systems for scheduling soft aperiodic task (to be discussed next). When scheduling aperiodic imprecise task is concerned, as anytime algorithm can be modelled as a soft and optional task, enhanced variants of server scheduling algorithms will be needed. To date this area of research has not yet been investigated.

3.5 Scheduling Soft Aperiodic Tasks with Spare Capacity

Servicing satisficing tasks is analogous to soft aperiodic task scheduling in real-time systems. Soft tasks can miss their deadlines when necessary whereas anytime algorithm can be stopped if needed. There is a significant body of literature related to scheduling soft aperiodic tasks in the presence of hard tasks. The approach, in general, is to schedule them using *spare capacity* that is unused by hard real-time tasks in the system. Various techniques have been developed in real-time systems. As will be discussed later, some of them may be exploited to facilitate anytime algorithms since their goal is to utilize available spare capacity to improves system utility. One emphasis among all these methods is to utilize spare capacity *safely* - a compulsory requirement which is not concerned by current RTAI research.

In practical real-time systems both periodic and aperiodic tasks are required. When non-critical aperiodic tasks are present in the system, the simplest way is to schedule them only when the processor is idle. This approach provides no guarantee for the responsiveness at all and thus another approach, known as *Periodic Server*, has been proposed. Average response time of aperiodic tasks is used as a primary metric of measuring system performance.

There are a number of different periodic servers. In general, a server algorithm is characterised by its period, priority among other tasks, capacity consumption rule and replenishment policy. Depending on these parameters, servers have different performance, run time complexity, implementation complexity and memory requirement. A server can be implemented as one of the hard tasks in a system scheduled by fixed priority algorithms such as RM and dynamic ones such as EDF; the server is referred as fixed priority and dynamic priority server respectively, depending on the algorithm used.

3.5.1 Background scheduling

Background scheduling is a simple intuitive way to service soft aperiodic tasks. Aperiodic tasks are only serviced when there are no other hard tasks ready and waiting in the system so the hard tasks will never be affected.

The advantage is that the implementation is simple. Only two queues are needed for scheduling; one with higher priority is for periodic tasks and another for aperiodic, which can be implemented by different algorithms. However, this can be adopted only when the aperiodic activities do not have stringent timing constraints and the periodic load is not high - the response time of aperiodic requests can be so long that many of them may miss their deadlines.

Note that the server scheduling algorithms below solve the above problem. However, generally they are not able to utilize all the spare capacity in the system. A background process is still useful for reclaiming all remaining resources even in the presence one or multiple periodic servers.

3.5.2 Fixed Priority Servers

When aperiodic tasks are not critical, we can execute them in background when the processor is not executing any hard tasks. The main problem is that there is no guarantee of responsiveness. When aperiodic tasks are critical, it can be incorporated into the rate-monotonic algorithm by the use of a periodic task, whose function is to service one or more aperiodic tasks. In addition, the scheduler has to maximize the processor availability for aperiodic tasks and at the same time minimize the response time.

The Polling Server (PS) is used to provide timely response for aperiodic tasks. A periodic task, called *Poller*, is used to service aperiodic requests. In particular, the ordering of aperiodic tasks does not depend on the scheduling algorithm for periodic tasks. If there is no aperiodic requests, the server suspends until the next period and the time originally allocated for the poller is not preserved for aperiodic execution but is used by perodic tasks. Server capacity is replenished at its full value every period. The problem is the incompatibility between the bursty nature of the aperiodic tasks and the periodic nature of the polling. When the server is ready there may not be aperiodic task to service; when there are aperiodic tasks the capacity of server may not be enough. Bandwidth preserving algorithms, like the deferrable server, priority exchange and sporadic server discussed below, are shown to be useful means to resolve this problem.

Deferrable Server (DS) was proposed by Lehoczky et al. [73, 113] to overcome the limitation in PS where aperiodic tasks arrives after the polling instant. It preserves the bandwidth allocated to the aperiodic tasks when there are no aperiodic tasks pending and thus improve average response time of aperiodic requests with respect to polling server. The algorithm does not incur more overhead than what the polling server does, with just a simple modification of the replenishment policy. This is an example of the importance of the server consumption rule and capacity replenishment policy.

However, DS does not behave like a periodic task. The consumption rule preserving the capacity in this way can cause what is known as a *back-to-back hit*, which is illustrated in Figure 3.3. The task set includes a deferrable server (capacity = 4, period = 8) and a task (WCET = 3, period = 10), together having total utilization equals 0.7 which is lower than the allowed utilization (about 0.83) using rate-monotonic policy. However, because the deferrable server preserves its capacity at time 16 and starts running at time 20, when there is aperiodic request. At time 24 it is again released and preempts task τ_1 to service aperiodic tasks. This causes extra interference to task τ_1 and it misses its deadline at time 30.

The behavior can not be accounted as usual periodic task's behavior and the maximum capacity (hav-

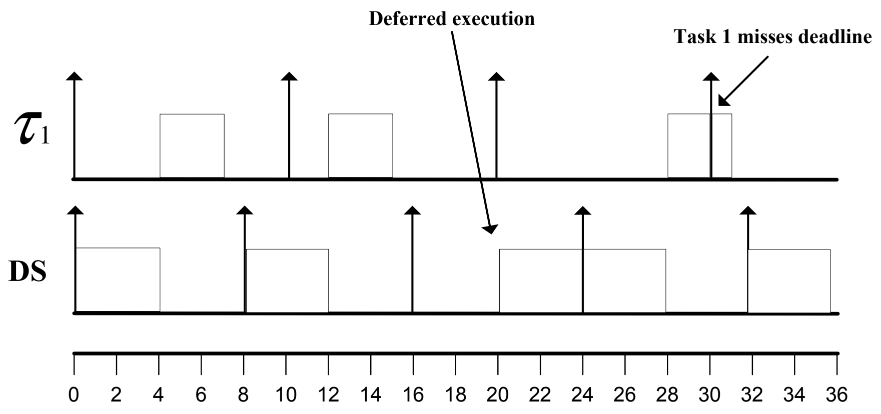

Figure 3.3: A task suffering a Back-to-Back hit when using Deferrable Server.

ing the same set of hard periodic tasks) achievable by DS is smaller than that of a poller. The least upper bound U_{lub} of a task set, with n number of tasks plus one deferrable server as the highest priority task, was shown to be [73]:

$$U_{lub} = U_s + n\left[(\frac{U_s + 2}{2U_s + 1})^{\frac{1}{n}} - 1\right] \tag{3.17}$$

where U_s stands for utilization of DS. As n tends to infinity, the bound converges to:

$$\lim_{n\to\infty} U_{lub} = U_s + ln(\frac{U_s + 2}{2U_s + 1}) \tag{3.18}$$

U_{lub} has the minimum value of 0.6518 when U_s = 0.186.

Deferrable Server normally runs at high priority; it has to run at the highest priority (shortest period according to RM) when one wants to analyse the schedulable utilization. There is no known schedulable utilization that assures the schedulability of the system when a deferrable server exists at arbitrary priority. When aperiodic tasks are firm, an acceptance test can be performed in order to guarantee their schedulability before they are admitted into the system.

Priority Exchange (PE) server is another technique introduced by Lehoczky et al. [73] for improving the response time of servicing aperiodic tasks achievable by polling. The PE algorithm also makes use of a periodic task which differs from DS in how the unused capacity is preserved. If there is no pending aperiodic task when the server is ready, it exchanges its capacity with the next ready periodic task with highest priority. When this exchange happens, the periodic task advances its execution and runs at priority level of the server; while the capacity of server is preserved at the priority level of the periodic task.

As shown in [73], the schedulability of the periodic task set is unaffected with a PE server. The total utilization U of the whole task set with n number of tasks plus a PE server as the highest priority task is:

$$U = U_s + n\left[(\frac{2}{U_s+1})^{\frac{1}{n}} - 1\right] \tag{3.19}$$

where U_s stands for utilization of PE. As n tends to infinity, the bound converges to:

$$\lim_{n\to\infty} U = U_s + ln(\frac{2}{U_s+1}) \tag{3.20}$$

Because total utilization $U = U_s + U_p$ where U_p is the utilization of periodic tasks other than the PE server, the whole task set is schedulable if:

$$U_s + U_p \leq U_s + ln(\frac{2}{U_s+1}) \tag{3.21}$$

So, all tasks other than the PE server is schedulable if:

$$U_p \leq ln(\frac{2}{U_s+1}) \tag{3.22}$$

Note that the above equation also applies to the polling server because both of them also behave like a periodic tasks.

Comparing Priority Exchange with Deferrable Server, PE provides better schedulability bound for periodic task set with a slight drawback in the worst case performance. However, Deferrable Server is less complex to implement because there is no need to keep track of the priority exchange among tasks.

Sporadic Server (SS), by Sprunt *el al.* [106], enhances the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set. Differed from DS and PE server that capacity replenishment is carried out every server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution. Let T_s be the period of SS and t_a be the time at which SS is active and start servicing aperiodic tasks. The next replenishment time t_r is set to be $t_r = t_a + T_s$.

In this way, SS behaves exactly like one or multiple periodic tasks; that is, it can be characterised by its processor utilization (execution time over period). The schedulability analysis for PE server is therefore also applicable to SS as well. However, performing schedulability analysis on firm aperiodic tasks is not easy because server capacity can be fragmented in a lot of small pieces of different size available at different times based on the replenishment policy. Consequently, one has to keep track of all the replenishments that will occur until the task deadline.

The bandwidth preserving algorithms mentioned above, when working under high loads, only provides performance similar to that of a polling server. Slack Stealing provides much better performance by using a different approach. Proposed by Lehoczky and Ramos-Thuel [72], rather than servicing aperiodic tasks by a periodic task, a passive task called *Slack Stealer* is used to monitor all the slack available from periodic tasks. Initial available slack are pre-computed offline and stored in a table for the least common multiple (LCM) of all tasks' period. At run time the current slack is computed from the table with $O(n)$ complexity, with the help of a set of counters to keep track of the slack at each priority level. The counters are decremented according to which tasks are executing and incremented by the values stored in the table at the next invocation of tasks.

Unfortunately, the table is of size n x N, n the number of tasks and N the number of tasks in the hyperperiod. The memory requirement can well be prohibitive for practical use even with a moderate task set. In addition, the effect of phasing and release-time jitters make this static approach difficult to be used in practice because pre-computed table will be invalid with only slight variations from the expected task's characteristics.

Davis et al. [42] introduced a method called Dynamic Slack Stealing (DSS) for fixed-priority scheduling that calculates slack at run time. Each time an aperiodic task enters into the system, the available slack at a particular priority level i is computed exactly by identifying the length of the busy period starting from a particular time t and the priority level i idle period given a particular starting time, again based on the response time analysis. This is a pseudo-polynomial time procedure (based on the number of tasks together with their deadlines and period) and is optimal in the sense that, at any given time, it can determine the maximum contiguous amount of spare capacity which can be used by non-hard tasks. Although the algorithm is more complex than the static approach at run time it allows handling of periodic tasks with release jitter or synchronization requirements which are problems in the static method because it does not depend on a pre-computed table, which, as the authors argued, can be applied to a wider class of problems.

It was originally thought that the Slack Stealer is optimal because it always advances all available slack as much as possible. Unfortunately, Tia et al. [116] showed that to minimize the response time of an aperiodic request, it is sometimes necessary to schedule it at a later time. In fact they proved that under *fixed-priority assignment* of periodic tasks, there is no algorithm (including dynamic slack stealer) that can minimize the response time of every aperiodic task. Furthermore, there does not exist any on-line algorithm that minimizes the "average response" time of soft aperiodic tasks. A weaker notion of optimality is thus defined for fixed-priority server scheduling and slack stealing algorithms are shown to be only optimal in the weak sense.

One thing that is worth noting is that the sporadic server has been adopted as a scheduling policy in POSIX (IEEE Std.1003.1d) [1] for its superior performance over the other server scheduling algorithms. However, Bernat and Burns [19] recently compared different types of server algorithms and showed that the server parameters selection and therefore the evaluation for sporadic server (SS) against deferrable server (DS) are not conclusive in previous studies by other authors. In previous analysis [106, 113], based on maximum possible utilization of the server and the system, SS was thought to outperform DS.

They also showed [19] that by using response time analysis rather than utilization based test [73], a different technique can be used in assigning server parameters. In particular, they argued that a DS can be modelled as a sporadic task with release jitter $J_i = T_i - C_i$. A weakness of utilization based test is that a same utilization $U_i = C_i/T_i$ can be achieved by various selections of C and T. Previous analysis assumed a particular value of period (highest priority using RM/DM) and therefore only certain value of capacity can be obtained. Depends on the task set, a higher utilization, both for DS and SS (which are almost the same), is in fact achievable by using response time analysis.

Server performance is found to be very sensitive to parameter selection. It was also found that a main factor that contributes to server performance is the server capacity. The original technique of determining server parameters, in particular server capacity by means of task utilization, is therefore inadequate.

Further, there is no optimal choice of server parameters that can be statically obtained beforehand and used subsequently with enough flexibility at runtime in dynamic environments. The best choice of server parameters is application dependent and parameters selection problem is hence non-trivial. More advanced techniques, both online and offline, are needed. The multiple server scheduling below addresses some of the inflexibility found in single server scheduling.

3.5.3 Multiple Servers and Capacity Sharing

Multiple servers with capacity sharing [20] is a generalization of single server scheduling. Multiple servers can better support aperiodic tasks with different timing constraints. In a dynamic system the characteristics of aperiodic tasks may vary so much that a single server, often using simple queueing rule such as FIFO, is inadequate. In particular, there is no mechanism in prioritizing different types of aperiodic tasks.

The capacity sharing mechanism ensures that when a server has used up its capacity it can share the capacity unused from other servers. With a simple sharing protocol Bernat and Burns [20] showed

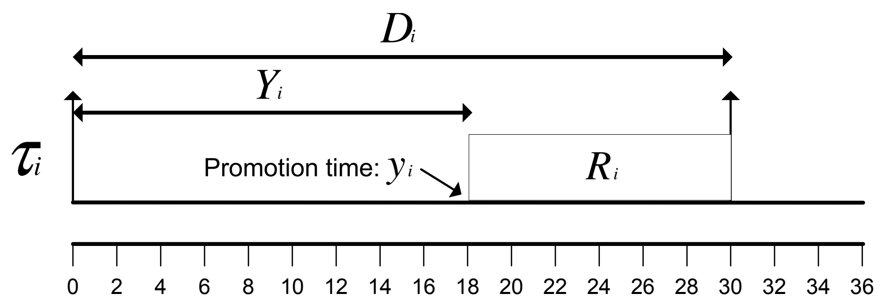

Figure 3.4: Promotion time of Dual Priority Scheduling.

that when scheduling tasks with varied loads, computation times and period distribution, multiple server scheduling outperforms single server scheduling in terms of average missed deadlines under heavy load, due to the prioritization of jobs in multiple servers. The single server algorithm treats all aperiodic the same and does not favour tasks with greatly varying deadlines.

As capacity sharing is advantageous in aperiodic tasks with varying deadlines [20], it can be useful in servicing RTAI tasks. Consider a planning algorithm which monitors a non-deterministic environment periodically (hard periodic task) and determine the deadline before which the planning must be performed. It then instantiates another (aperiodic) task to perform the actual planning. If there is only one single aperiodic server with other jobs waiting, it may not be successfully scheduled with a short deadline. Multiple servers are more flexible in situations like this.

The authors also noted that the capacity sharing can be restricted to only a subset (cluster) of all servers [20]. The circumstances for which clustering should be performed are not known. An interesting direction to look at is the prospect of performing dynamic clustering of servers based on the dynamic task requirements. Moreover, there is still the question of how multiple servers can be made to support imprecise computation.

3.5.4 Dual Priority Scheduling

Even with dynamic slack stealing, calculating slack is computationally expensive. Apart from using approximate method for calculating slack [39], Davis and Wellings presented a superior mechanism called Dual Priority Scheduling (DPS) [40, 43] to tackle the complexity problem of calculating slack for servicing soft and optional tasks.

DPS takes a different approach of scheduling aperiodic tasks compared to that of periodic servers and slack stealer. In DPS, a promotion time is defined for each hard periodic tasks calculated using their

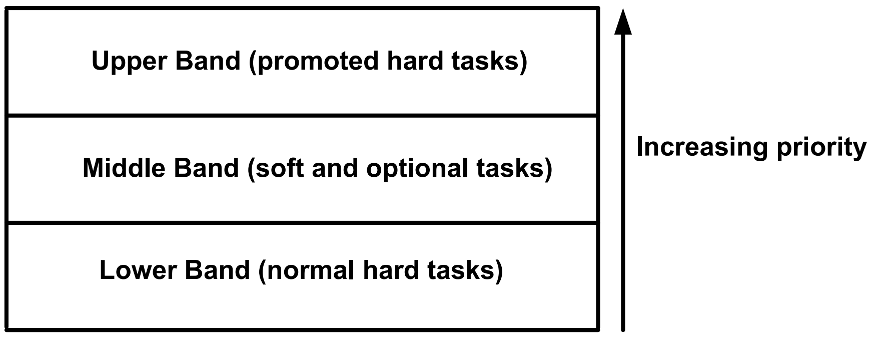
Upper Band (promoted hard tasks)
Middle Band (soft and optional tasks)
Lower Band (normal hard tasks)
Increasing priority

Figure 3.5: Promotion hierarchy of Dual Priority Scheduling.

maximum response time to allow spare capacity to be advanced as much as possible. Suppose R_i is the response time of task τ_i obtained by using response time analysis and t_i be the task release time. The latest promotion time y_i is set to $y_i = t_i + (D_i - R_i)$ where D_i is the deadline of task τ_i (Figure 3.4).

Hard tasks execute at either the upper or lower band. Upon release, each hard task runs at lower band by default. The promotion time of a hard task is set by an offset $Y_i = D_i - R_i$ from release, the task is promoted to upper band if by the promotion time it has not finished. If no aperiodic task is pending the hard task will be executed immediately. Soft aperiodic tasks run at the middle band and they can preempt any hard tasks at the lower band without jeopardising their schedulability (see Figure 3.5).

Although it is not optimal in the sense of stealing every possible capacity, the performance of dual priority scheduling is shown to perform better than the Improved Priority Exchange server, and very close to Slack Stealer. With only $O(log\ n)$ execution time overhead it is much preferable to slack stealing [43].

The authors also incorporated release jitter, arbitrary deadlines and resource access protocol (priority ceiling protocol) into the promotion time calculation. DPS provides an efficient way of providing spare capacity for scheduling optional and soft task in an earlier time. It is a useful technique for scheduling and has been adopted in other scheduling contexts. Examples include weakly hard real-time scheduling [18], real-time multiprocessor scheduling [15] and energy-aware computing [86].

However, servicing imprecise computation with dual-priority priority scheduling results in incompatability. Consider a hard task with unbounded optional component where the mandatory subtask is guaranteed by the response time analysis. Due to the fact that dual-priority scheduling tries to postpone the computation of hard tasks as late as possible, which takes maximum fixed-priority preemptive interference into account. The time that is available for executing the unbounded component is restrained.

In fact, due to the precedence constraint of the mandatory and optional components, it may be more justified to execute the mandatory components as *early* as possible, albeit the scheduler may also be trying to minimize the average response time of aperiodic tasks as well as scheduling as much imprecise optional parts as possible . More studies are needed if dual-priority is to be used in scheduling a very mixed type of computation, where tasks can be defined by any combinations of the following properties: hard/soft/firm-time, periodic/sporadic/aperiodic, error-minimizing/reward-maximizing and bounded/unbounded imprecise computation.

3.5.5 Resource Reclaiming

There is another kind of spare resource rather than the unused capacity by non-hard tasks, which is dynamically available at runtime, due to hard tasks not executing up to their WCET and sporadic tasks not arriving at maximum possible rate called *Gain Time*. To utilize such capacity, care must be taken because if tasks are allowed to execute for this extra time, lower priority tasks may suffer longer interference not included in the feasibility test carried out.

Gain time can be identified at various stages of task execution. One approach is by examining task input parameters [111] because it may have significant effect on the execution path of a task and thus its running time. Another way is to monitor task's execution by the use of Gain Points [5] or Milestones [48]. In the milestones method, execution is divided into different stages and if particular stage finishes before the worst-case time, gain time can be identified.

Bernat et al. proposed a method to utilize gain time by retrospectively rewriting task's execution history [17]. With this scheme, any gain time is preserved and re-allocated iteratively for lower priority tasks in such a way that it will not affect system schedulability. Normally, any gain time that is unused by other tasks have to be thrown away at next replenishment. By the mechanism of history rewriting (at the end of task's period rather than at the time the task finishes), the spare capacity is allocated to the next lowest task; if there is still capacity left the procedure goes on to the next lowest priority task. It was also shown that the scheme works well with capacity sharing [20], response time analysis and existing server scheduling techniques, such as the Deferrable Server algorithm.

Assumed that usually there is no advantage for finishing hard time earlier and their WCET estimation is accurate, this scheme seems not particularly useful for sharing gain time for other hard periodic tasks. The authors [17] have considered using gain time for facilitating imprecise computation. However, in their example, hard tasks and unbounded tasks are arranged to be interleaved so that gain time can be shared to unbounded tasks immediately. This arrangement imposes restriction on task priority assignment and the construction of system.

Another issue with slack reclaiming for soft tasks is that the amount of slack reclaimed at a finer scale may not be very useful for, say, deliberation tasks, which may work at much greater time granularity. It may turn out that the cost of slack reclaiming for deliberation is not justified in terms of the possible gain of utility.

Sophisticated gain time reclaiming mechanism may not be useful in terms of scheduling optional tasks because: 1) in general, there is no utility gain in finishing hard tasks early so methods for reclaiming gain time for hard tasks are not needed; 2) gain time should be used to facilitate satisficing, and independent scheduling algorithms can be employed particularly for optional tasks.

For example, the dual priority scheduling naturally provides a notion of resource reclaiming built into the scheduling mechanism and specialized algorithm can be used in the "middle band" of dual priority scheduling for deciding how to share the capacity among aperiodic and optional tasks. There is no extra overhead to reclaim gain time and more importantly, to avoid the problem of considering tradeoffs in gain time granularity. The scheduling decision is up to the policy defined in the middle band and adaptive behavior may be achieved by utilizing different algorithms under different situations.

3.6 Multiple Applications and Hierarchical Scheduling

Open system architecture, first proposed by Deng and Liu [46], is a two-level scheduling framework within which different applications can have their own scheduling algorithms, independent of the scheduling implementation of the underlying system. An application can use the most suitable scheduling policy to itself. Most importantly, schedulability and real-time constraints can be determined locally within each application, without concerning other applications running on the same system.

Nowadays, complex systems are composed of different applications written independently with possible arbitrary combination. As mentioned, more recent RTAI research direction has been building RTAI application as a subsystem or module which is part of other larger systems [53]. The traditional real-time scheduling such as the RM and EDF algorithms, although proved to be optimal with certain assumptions on task attributes, can not accommodate all the special needs required by various applications. Subsystems will not all work well with a single scheduling algorithm at the system, each of them may require a scheduling algorithm that is most suitable to its task nature and timing constraints. Hence, there has been a growing attention in studying hierarchical scheduling in the literature.

Two-level scheduling scheme [46, 47, 81] provides temporal isolation to individual applications running on a same processor where each application can contain arbitrary number and types of tasks. Schedulability of tasks in individual application can be guaranteed independently by using an infinitesimally fine-grain time slicing method. Of course, "infinitely" small quantum is not possible because the context switch overhead will become dominant and cause *thrashing*[3]. The system scheduler essentially emulates a "slower" processor for each application. Each application is scheduled by a server with suitable bandwidth determined by a function of required budget by the application. The OS scheduler uses a constant bandwidth server or a total bandwidth server to service each application.

Deng et al. [46, 47] devised sufficient conditions - the correct suitable server budget size and replenishment policy - that must be satisfied when guaranteeing real-time performance of higher-level applications. Predictable applications are ones that schedule tasks nonpreemptively, in a clock-triggered manner, or preemptive periodic task set with appropriate modifications in server's budget and deadlines, whereas unpredictable applications are those scheduled according to preemptive, priority-driven algorithm and contain aperiodic tasks and/or periodic tasks with release-time jitters.

Building upon the work of Deng and Liu, Kuo and Li [70] studied the same architecture, but using fixed-priority scheduling. In the architecture, they used a system scheduler with RM priority assignment and a sporadic server for each application because constant/total bandwidth server is not compatible with the RM algorithm. They presented the corresponding schedulability test, based on available and required utilization, for the cases where individual applications schedule their own tasks based on RM and EDF. However, there is a restrictive assumption that for each application, its corresponding server's period has to be the greatest common divisor (GCD) or a divisor of the GCD of all the tasks in the application.

Using fixed-priority scheduling as the system scheduler, another approach is to use response time analysis for checking schedulability of applications. As discussed before, response time analysis has many merits. Saewong et al. [98] showed a response time analysis for two-level systems within which an application is scheduled by either a deferrable server or a sporadic server. Lipari and Bini [79] have proposed an alternate formulation of response time analysis by using an availability function representing the available capacity made by a server. However, as argued in [38], the schedulability is sufficient and

[3]By which the system is only busy executing context switch routines rather than user tasks.

not necessary because a pessimistic assumption of the worst-case capacity availability of a server was used. Lipari and Bini [79] also considered the problem of server parameter selection, such as server period and capacity, which is, however, only concerning a single server without considering its effects globally.

Davis and Burns [38], based on the work mentioned above, studied fixed-priority preemptive scheduling as applied at both system level and local application level. A set of applications is scheduled by fixed-priority preemptive scheduling in the system level; local tasks belong to a single application are run with a periodic server or deferrable server. A response time schedulability test was presented for accounting interference that a task may suffer from other applications in the system.

One interesting insight is that, because tasks in an application can only be executed when the server is released in the base system, periodic server outperforms deferrable server in such hierarchical framework. When the performance measure is the task schedulability of application rather than aperiodic task's responsiveness, harmonic server periods enable some tasks to be bounded to the release time of their server, for which a simple poller gains advantage.

They also investigated the optimal selection of server parameters and show that there are dependencies among servers and locally optimal selection does not result in global optimality. That is, it is a holistic problem where global searching techniques may have to be employed offline for finding the best parameters selection. It was also shown that global resource sharing across different applications can not be easily analysed.

There are also attempts in generalizing hierarchical scheduling. Regehr and Stankovic [90, 91] proposed a more general architecture by which unlimited levels of hierarchy can be constructed with a main focus on soft real-time guarantee in general purpose operating system (GPOS), called Hierarchical Loadable Scheduler (HLS). They categorized different scheduling algorithms and the types of real-time guarantees they can provide, in order to impose restrictions on the scheduler composition hierarchy. For example, a fixed-priority preemptive scheduler can provide timing guarantee for a time-sharing scheduler (in GPOS); but not vice versa.

Feng and Mok [50] showed that in guaranteeing tasks' schedulability, global knowledge of the entire task set is not necessary in hierarchical scheduling. When the notion of virtual resources with bounded delay and variation is introduced, there is no need for global task analysis. The main feature of this work is the abstraction of virtual resource augmented by a notion of the rate of service provided by the underlying layer and the specification of the maximum delay suffered. Real-time components can then be composited in such a way that they only need to consider the virtual resource provided. The advantage is that feasibility test can be performed locally which is clearly desirable in open dynamic environment where many properties of tasks are unknown in advanced.

Despite that the work of multi-level hierarchical scheduling have addressed some real-time guarantee inheritance and scheduler composition problem, the servicing of aperiodic and further, imprecise computation within a single application in hierarchical systems (even just two-level ones) has not been studied at any depth. Further investigations are also needed to derive sufficient, and if possible necessary, conditions for guaranteeing applications consisting anytime algorithms and multiple methods because of their different task nature.

The hierarchical architecture is of great importance for open, adaptive systems. However, there is little work in the literature on choosing server parameters that are suitable for fulfilling particular application requirements. As shown by Davis and Burns [38], there are situations where a simple poller outperforms deferrable server and sporadic server. A more advanced technique for parameter selection should be developed which takes into account task's characteristics in different applications. Such technique can benefit the schedulability of the system globally, especially in situation where parameter adjustments of existing servers are necessary for accommodating new applications. Interested readers can refer to [29] and the references therein for more details on hierarchical scheduling in composite systems.

3.7 Practical Issues: Programming Language and System Support

Programming language concerns about the software engineering aspect while operating system regards the operational issues of systems. Yet, they are closely coupled and are both essential to any successful systems.

When Liu et al. first proposed the imprecise computation work, they used C++ augmented with a set of language extensions called Flex [67], which is included in a system named Concord, for supporting milestones method. Concord [82] is a system that provides the programming language primitives and system support for imprecise computation. Based on a client-server model, it allows the programmer to specify the intermediate result variables to be recorded and define the error indicators for monitoring the intermediate result.

A more comprehensive Imprecise Computation Environment (ICE) was developed later by Hull et al. [64]. Also based on a client-server architecture, ICE is implemented on top of Real-Time Mach (RT-Mach). The kernel is significantly modified for adding the NORA online scheduling algorithms for imprecise computation [103]. In ICE, there are also features for supporting dependability through incorporating fault-tolerance into the imprecise real-time systems. The server supports a user-directed checkpointing mechanism for imprecise tasks, where rollback recovery is possible.

Gregory and Lin have proposed two implementation packages for realizing imprecise computation in a previous version of Ada [55]. Though in the process they identified several problems related to Ada such as the time-expensive rendezvous, nonpreemptiveness of task model and the potential much longer delay using the *delay* statement in Ada. Audsley et al. [7], in investigating how to integrate unbounded software components into hard real-time systems based on response time analysis, where a *"Prologue-Optional-Epilogue"* task structure was proposed, showed the model can be implemented in Ada 95.

An interesting area to look at will be the real-time scheduling support for common (functional and logical) languages used in AI, such as Common LISP, Prolog and Progol. They are very popular languages for constructing planning, searching and learning algorithms [96]. Profiling is a common practice for measuring the WCET of tasks in actual system; the data is then used to perform offline feasibility test. A suitable CASE tool for generating algorithm's performance profile, schedulability verification and possibly automatic code generation, will be of great prospect.

Stankovic and Rajkumar have presented an extensive review in real-time opearating systems (RTOS) recently [109]. RTOS has been changing from the traditional small, proprietary kernel to more component-based and QoS-based kernel, from the traditional requirement of providing hard and soft guarantee to new paradigms that provide admission control, resource reservation management and reflection. Not surprisely, to add real-time capabilities into existing systems, various extensions have been applied to commercial and research type operating systems, such as RT-Unix, RT-Mach, RT-Linux. In particular, among other open source RTOS, Linux has been used more extensively in real-time research due to its availability of source code and community support.

In fact in small proprietary kernel, all tasks may be analysed by a simple schedulability test because the task set is small and all the task parameters are known in such "closed" system. For general purpose operating system augmented with real-time capabilities, such as RT-Linux, the architecture resembles the two-level framework shown by Deng and Liu [45]. First, the system is open - new application can be added into the system; and second, each application can be supported by a scheduling algorithm that is most suitable for itself. Real-time systems are becoming more heterogenous with mixed type of real-time requirements.

Many new scheduling analysis, such as the hierarchical scheduling mentioned earlier, is performed by theoretical derivation and/or via simulation, without real implementation in a RTOS, due to the efforts required in constructing a new scheduling algorithm in an OS. Consequently the practical aspects of these algorithms are not known and they are also generally not available for use in real systems. Increasing efforts have been witnessed in adding more efficient and flexible scheduling in RTOS. The followings serve as examples to show the general trend in RTOS but do not claim to be comprehensive.

By using a generic scheduling framework, S.H.a.R.K. (Soft and Hard Real-time Kernel) provides a dynamically reconfigurable kernel architecture for flexibly constructing, integrating and evaluating different scheduling algorithms [51]. The kernel itself is fully modular regarding scheduling algorithms and resource access protocols. Scheduling policy can be isolated from the core of the kernel. New scheduling algorithm can thus be written without concern of the kernel but the algorithmic scheduling property. Many well known algorithms, such as RM, EDF, have already been implemented.

Burns and Bernat [24] have proposed a scheduling framework that can be efficiently implemented in RTOS. The framework supports a mixture of hard and firm tasks and provides low implementation overhead. In more open environment where task properties are not all known a priori, as the authors argue, efficient scheduling and admission tests are needed. By using a threshold, value-based acceptance test, the authors showed the time complexity of scheduling and admitting firm tasks can be kept low ($O(log\ n)$) and are easy to be implemented in real systems.

Another scheduling framework is the FIRST Scheduling Framework (FSF) [3] which stressed on the scheduling abstraction independent of implementations. Also notwithstanding the low utilization in traditional hard real-time systems, the framework comprises a set of standard scheduling algorithms such as table driven, fixed priority and EDF. A main feature of the framework is the specification of real-time requirement in a service contract, where the QoS is negotiated between applications and the supporting system. FSF is essentially a set of API so that it can be independent from implementations. The library provides many different and complex services, from simple budget control to synchronization primitives, hierarchical scheduling (two-level), reclamation, shared objects and distribution. It has been implemented in two real-time kernels: S.H.a.R.K. [51] and MaRTE OS [92] where results are shown to be satisfactory.

Flexible Real-Time Linux (FRTL) [115] aims to provide support for imprecise computation in Real-time Linux (RT-Linux), which is a general purpose operating system (Linux) augmented with real-time capabilities. In its scheduling framework, a task is composed of a sequence of mandatory and optional components where there executions are separated in two levels. All mandatory tasks are executed in the base scheduler and are guaranteed using the response time analysis [10, 8, 6] where all optional components and non-real-time tasks are scheduled using any slack available, identified by the dynamic slack stealing algorithm by Davis et al. [42] [4].

The framework provides greater freedom than those offered in the previous approach in that it does not restrict the number of mandatory and optional parts of a task. More importantly, the overhead of the FRTL implementation, such as interrupt handling, context switch time and systems calls, is evaluated and included into the response time equation to provide accurate analysis of the whole system. They showed that FRTL can be efficiently implemented in a regular PC.

Regarding RTAI, Stankovic et al. [112] demonstrated the collective supports, in terms of specification language, programming language, compiling tools and the kernel, that the Spring OS can provide in order to support complex real-time applications. In the case study, a manufacturing testbed involving both real-time systems and AI automated reasoning is concerned. There are two levels of scheduler: the higher level a scheduler, called the AI planner, is used for making decisions of what order to service and when to service them; whereas at the lower level scheduler is responsible for real-time scheduling and the feasibility of servicing particular order.

The study showed that:

- the separation of, and the cooperation between AI reasoning system and real-time systems are useful and applicable in real world problem;
- the success of RTAI applications depends on extensive support provided by good programming environments and runtime systems; and
- there are new types of task interdependencies that are not usually present in real-time systems and the interface between the two systems is essential in resolving potential conflicts.

[4] Note that a much lower scheduling overhead is possible by using dual-priority scheduling [40, 43].

3.8 Summary

RTAI applications appear to require most of the existing technologies reviewed above, but clearly they need to be in a more integrated fashion.

The basic problem of guaranteeing hard periodic tasks in real-time systems has been solved. In particular, the response time test is flexible and can be extended to satisfy different requirements such as resource access blocking time, release jitters, offsets, etc [6, 27]. Meeting hard timing constraints will continue to be the primitive objective in real-time systems.

The imprecise computation provides flexibility and a formal mechanism for performing tradeoffs and graceful degradation in real-time systems during, for example, transient overload.

The use of periodic servers for scheduling aperiodic tasks has become more common in real-time systems. Owing to simple queueing policy and assumptions in single server scheduling, aperiodic tasks with different timing characteristics, which may well be schedulable with other different strategies, can miss their deadlines. The use of multiple severs and capacity sharing comes into light in this situation.

Due to the similarity of soft tasks and imprecise computation, the work in scheduling soft aperiodic tasks by making use of unused capacity, especially by means of periodic server, is applicable in facilitating satisficing techniques. For example, a periodic server may be extended to service both aperiodic and imprecise computation. The multiple server and dual priority scheduling should also be investigated for the same rationale.

Current real-time scheduling is moving towards investigating more open and dynamic systems [99], which coincide with the type of RTAI systems of interest. As more adaptive and flexible scheduling is demanded, there will be more research on multiple server scheduling and hierarchical scheduling. However, no single scheduling framework is likely to satisfy all application requirements, especially if they are unknown in advance. A mechanism by which scheduling algorithms and parameters can be adjusted dynamically looks to be a promising field of research for supporting deliberative agents.

To sum up, in this chapter related works in real-time systems, from a perspective of what deliberative agents need, have been reviewed. It is shown that the requirements from RTAI and the inadequacies of corresponding supports in current real-time scheduling require extended research. In the next chapter, new results in scheduling theory for imprecise computation based on the P-O-E task model are presented.

Chapter 4

Fixed Priority Preemptive Scheduling for Imprecise Computation

Consider the Robocup case study mentioned in Chapter 1 where an anytime planning algorithm[1] [44, 69, 125] of which the initial part requires hard guarantee for constructing a minimum acceptable plan. The algorithm then refines the quality of the results whenever there are available resources. At some point it has to stop to meet its timing requirement where the final result may need to be written back to a shared memory area, an I/O device or sent over a network (e.g., in a cooperative multi-agent system) and wait for the corresponding result acknowledgement, which with real-time requirements may require a particular real-time access protocol for guaranteeing schedulability.

Using the traditional "Mandatory-Optional" task structure [83], the final part of the computation is not guaranteed and may not finish before its deadline. Moreover, the optional components of the planning algorithm can be unbounded while the optional part of the original model is however, characterised by an error function, which assumes that when it is executed to its completion, the task is said to produce *no error*. Anytime algorithms in this regard introduce a difficulty for Liu's model since there is no clear bound that, if reached, a perfect result will be produced. In addition, the hard part of computation required *after* the optional part can not be included in the model.

The "Prologue-Optional-Epilogue" (P-O-E) task model proposed by Audsley el al. [11, 7] is a generalisation of that of Liu et al as mentioned and can accommodate tasks of such structure. The authors devised schedulability tests, based on response time analysis with task offsets [8, 117], for guaranteeing

[1]Analogous to the "milestone method" in imprecise computation.

schedulability of the prologue and epilogue (mandatory) tasks. However, in examining the proposed schedulability tests in [11, 7], as shown in later in this chapter, it was found that the procedure may not calculate the actual worst-case response time due to a flaw in the suggested offset transformation. The problem was rectified by considering all possible critical instants introduced by the different combinations of task's offsets where the time complexity1 is exponential to the number of imprecise tasks concerned. Furthermore, it is shown that, under Deadline Monotonic (DM) priority assignment, which is also assumed in [11, 7], one of the tests is not necessary, resulting in simpler schedulability analysis.

The remainder of this chapter is organised as follows: in the next section the task model of concern is first defined, where the schedulability analysis from [7] for tasks with offsets is restated. The existing problem of inaccurate response time calculation is then presented and a solution is proposed. This is followed by a proof on how the original tests can be simplified when deadline monotonic ordering is employed.

4.1 Task Model

The task computation model is based on that of [7] (also in the previous chapter), with a slight change in task notation.

A general computation can be modelled by five components:

$$I, C1, X, C2, O$$

where I and O stands for task input and output, $C1$ and $C2$ for mandatory computation, and X optional computation.

Note that the I/O time may depend on the processing speed of the corresponding devices and the access to the devices can be bounded by using a particular resource access protocol. Milestone methods (including anytime algorithms), sieve functions, and multiple versions[2] can then be represented with different arrangements from these parts [11, 7].

A set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ represents all tasks in the system. A task τ_i is represented by 3 different components; the prologue (τ_i^p), epilogue (τ_i^e) and optional (τ_i^o) component where:

[2]Also called multiple methods in other contexts.

$$\tau_i^p = \{I, C1\},$$
$$\tau_i^e = \{C2, O\},$$
$$\tau_i^o = \{X\}.$$

This is denoted as the *"P-O-E"* task structure. An ordinary periodic task is just an imprecise task without the optional and epilogue part. The P-O-E structure is a generalisation of the Mandatory-Optional structure because the latter can be represented by a P-O-E part without the epilogue part.[3]

Each task τ_i has the following parameters:

- worst-case execution time (WCET) C_i, where C_i^p is the WCET of τ_i^p, C_i^e the WCET of τ_i^e and $C_i = C_i^p + C_i^e$;
- deadline D_i. The deadline of task task τ_i, which also applies to both τ_i^p and τ_i^e;
- period T_i. The period of task τ_i, where τ_i^p and τ_i^e share the same period length;
- offset O_i. Whenever applicable, the offset of task τ_i causes the task to be released after the period, with the length O_i. O_i^p and O_i^e represent the offsets for both τ_i^p and τ_i^e from their periods.

Both τ_i^p and τ_i^e need to finish execution before their deadlines, otherwise the task τ_i is said to have missed the deadline. For clarity, if a task is mentioned without direct indication that it is imprecise, it is assumed to be an ordinary (non-imprecise) task. When we mention the *partner task* τ_k of a task τ_j, if τ_j is the prologue then τ_k is the epilogue from the same imprecise task and vice versa.

A level-i busy period refers the execution duration of a task τ_i (with priority i) from the instant it is released to the instant it finishes the execution, during which the system is busy executing τ_i or higher priority tasks. The longest busy period of the task is its worst-case response time and the corresponding release instant will be the *critical instant* [80]. A schedulability test determines if a task set is schedulable given a priority assignment ordering and is called sufficient if all tasks passing the test are indeed

[3]The authors acknowledge that the P-O-E structure can also be represented by two end-to-end Mandatory-Optional imprecise tasks [49] where any error produced in the first task will not result in any *input error* to the second - they merely have a precedence relation, with the optional component of the second task discarded. However, it is the author's belief that the P-O-E structure is conceptually more concise and general for a *single complete* computation. Representing the P-O-E that way necessarily complicates the required system support and confuses system designers from a system engineering perspective.

schedulable. It is called necessary if any task sets that fail the test is actually unschedulable. A test is exact if it is both sufficient and necessary.

4.2 Schedulability Analysis

The primary objective of a real time system is to guarantee all ordinary hard tasks, as well as the mandatory parts (prologue and epilogue) of imprecise tasks, will meet their deadlines.

In a fixed priority preemptive system, the schedulability of the system can be easily found by using the response time analysis by completely omitting the optional task. If the epilogue part is immediately executed once the prologue part is finished, from a scheduling point of view, the two parts can be treated as one single task.

Assume that the deadline of a task is equal to or smaller than the task's period, the worst-case response time R_i of a task τ_i, as described in the previous chapter, can be stated by:

$$R_i = C_i + B_i + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \tag{4.1}$$

where $hp(i)$ stands for all higher priority tasks of task τ_i. The term B_i represents the maximum blocking time of accessing a shared resource. A bound can be found using a particular resource access protocol, such as the Priority Ceiling Protocol [101]. In that case B_i represents the maximum execution time of any lower priority tasks of the task τ_i.

The equation can be solved by iteratively evaluating the following recurrence relation [8], which can begin with $r_i^0 = 0$ or $r_i^0 = C_i$, the worst-case response time r_i of a task τ_i is found when $r_i^{n+1} = r_i^n$ or $r_i^{n+1} > D_i$, in that case the task set is unschedulable (see [66] for proof of convergence for task sets with $\leq 100\%$ processor utilization).

$$r_i^{n+1} = C_i + B_i + \sum_{j \in \mathbf{hp}(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j. \tag{4.2}$$

4.2.1 Making Use of Task Offset

The above test, however, is ineffective in facilitating optional components since it makes no attempts to schedule them but just pure schedulability of mandatory tasks. Since there is a precedence relationship between τ_i^p and τ_i^e, to make rooms for execution of optional components, an intermediate deadline S_i

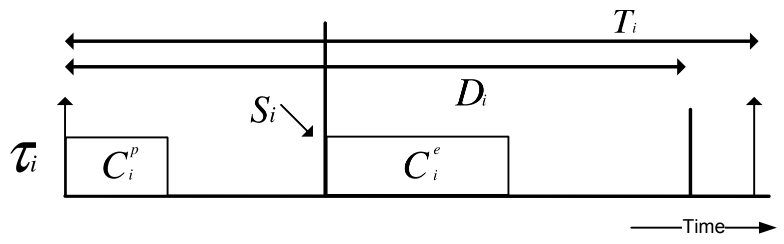

Figure 4.1: Intermediate deadline assignment for imprecise task.

can be used to separate the two tasks [7], effectively setting an offset to τ_i^e relative to the release time of τ_i^p.

A reasonable intermediate deadline S_i (which divides the slack equally for the two tasks) is (see Figure 4.1):

$$S_i = \frac{D_i - (C_i^p + C_i^e)}{2} + C_i^p. \tag{4.3}$$

As mentioned, the period of each mandatory task, T_i^p and T_i^e, is equal to T_i. The offsets and deadlines of the tasks are modified as:

$$\begin{aligned} O_i^p &= 0, \\ O_i^e &= S_i, \\ D_i^p &= S_i, \\ D_i^e &= D_i - S_i. \end{aligned}$$

The selection of S_i is somewhat arbitrary. However, setting it at an earlier time will make it more difficult to schedule the prologue task τ_i^p because it now has a shorter deadline. On the other hand, setting S_i to a later time will decrease the deadline of the epilogue task and may affect its priority and schedulability. More discussions are given on issues of selecting S_i in the next chapter.

Applying the equation (4.2) again with the modified deadlines results in a less pessimistic analysis because the two mandatory tasks of each imprecise task are now treated individually. However, an even less pessimistic test can be obtained by making use of task offsets.

4.2.2 Task Priority Ordering

Neither Rate Montonic (RM) [80] nor DM [76] ordering is optimal when arbitrary offset is allowed [117]. Here DM ordering is assumed. According to the modified deadline mentioned above, each task is assigned a distinct priority - the smaller the deadline the higher the priority. For tasks with same deadline value they are each assigned a distinct but adjacent priority.

For the purpose of performing schedulability tests, each imprecise task will be individually represented by the two mandatory tasks (τ_i^p, τ_i^e), each being treated as a single ordinary hard task in the tests. Therefore, if there are n imprecise tasks and m ordinary tasks in the system, there are in total $2n + m$ tasks in the schedulability analysis.

Here a shorthand τ_j is introduced which represents a task with priority level j (every τ_i^p and τ_i^e is now represented by a task τ_j with a distinct priority j), where $1 \leq j \leq 2n + m$ and the lower the index the higher the actual priority.

4.2.3 Schedulability Analysis with Task Offsets

Consider a task τ_j. The only tasks that can increase its response time are the higher priority tasks. When the τ_i^p and τ_i^e pair both have higher priority, to account for the worst-case, assuming a critical instant when all tasks are released at time 0, the task in the pair with the greatest computation time is assumed to be released together with τ_j at time 0, and the task with lower computation time in the pair is assumed to be released at an offset from 0.

More specifically, when $C_i^p < C_i^e$ the offsets of them are transformed as follows:

$$O_i^p = T_i - S_i,$$
$$O_i^e = 0.$$

When only one of the pair has higher priority than τ_j, it is assumed that it is released at the same time with τ_j. For instance, if τ_i^e is the only higher priority task in the pair then its offset is reassigned as:

$$O_i^e = 0.$$

The above only accounts for higher priority tasks, the partner task of τ_j, let's say τ_k, also needs to be considered. If the partner is of lower priority, it obviously cannot affect τ_j. A less pessimistic analysis can then be obtained by restating equation (4.2) as:

$$r_j^{n+1} = C_j + B_j + \sum_{x \in \mathbf{hp}(j)} \left\lceil \frac{r_j^n - O_x}{T_x} \right\rceil C_x. \tag{4.4}$$

However, if τ_k is a higher priority task than τ_j, because it is possible that it can cause other tasks with intermediate priority to delay their response times, the response time of τ_j can also be affected indirectly.

According to the previous analysis [7], one must examine two priority level j busy periods. The first one assumes that τ_j is released at time 0 and ignoring τ_k. Another assumes that τ_k is invoked at time 0 with τ_j releasing at an offset of O_j. And finally the maximum of the two busy periods is taken.

Let w_j denote the length of a priority level j busy period. The length of the first busy period can be found using the recurrence relation shown below:

$$w_j^{n+1} = C_j + B_j + \sum_{x \in \mathbf{hp}(j) \wedge x \neq k} \left\lceil \frac{w_j^n - O_x}{T_x} \right\rceil C_x. \qquad (4.5)$$

Iterations follow that of equation (4.2). Note that possible interference from τ_k is excluded and the response time, r_j is equal to the busy period w_j.

For calculating the second busy period, set $O_j = T_i - S_i$ if τ_j is the prologue task of process τ_i, and $O_j = S_i$ otherwise. The busy period w_j is found as follows:

$$w_j^{n+1} = \left\lceil \frac{w_j^n - O_j}{T_j} \right\rceil C_j + B_j + C_k + \sum_{x \in \mathbf{hp}(j) \wedge x \neq k} \left\lceil \frac{w_j^n - O_x}{T_x} \right\rceil C_x. \qquad (4.6)$$

In the above equation, the term C_k represents the interference due to one invocation of τ_k because τ_k and τ_j share the same period. The actual busy period is $w_j - O_j$ after the value w_j converged, since task τ_j is assumed to be released with an offset O_j.

The worst-case response time r_j corresponds to the maximum of the two busy periods found in equation (4.5) and (4.6), when its partner task is having a higher priority. And schedulability can be easily known by comparing to its deadline D_j.

4.2.4 The Problem with Offset Transformation

The procedure above restates the main results in [7] and now the problems associated with the offset transformation are presented. In essence, it is argued that, when considering the worst-case interference of a task from an imprecise task where both prologue and epilogue have higher priorities, merely transforming the one with greater computation time to be released at the same time with the task concerned is not adequate. That is, the transformation does not resemble the critical instant for the task.

Consider a task τ_x and an imprecise task τ_i with prologue τ_j and epilogue τ_k where τ_i has a relatively

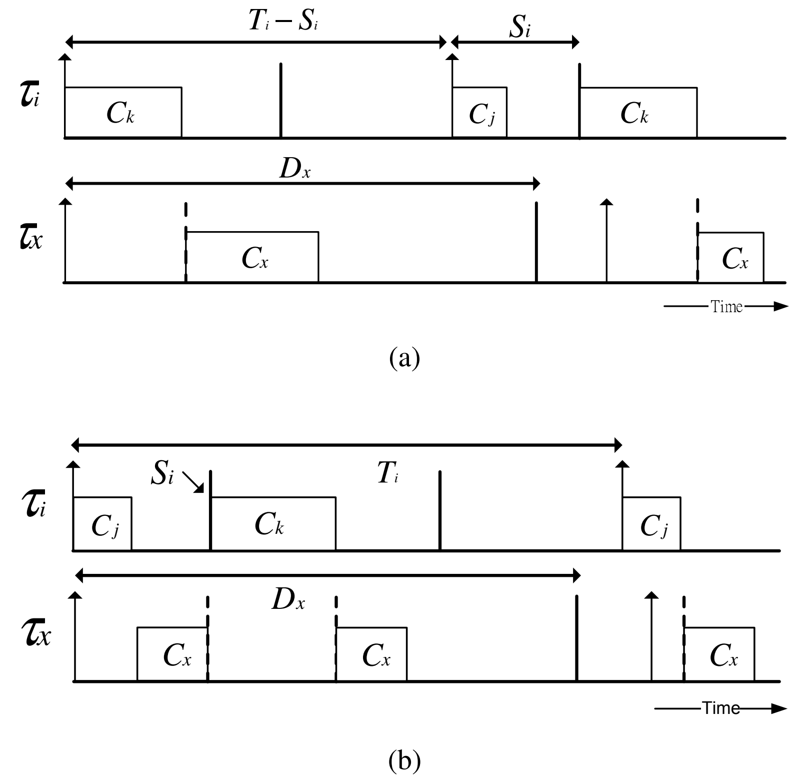

Figure 4.2: Illustration of the offset transformation problem.

short deadline compared to its period (Figure 4.2). With $C_j \leq C_k$, perhaps intuitively, τ_j will suffer the longest interference when τ_k is released at the same time with τ_x. However, following the offset transformation rule, task τ_i will have an offset set as $T_i - S_i$, where S_i is the intermediate deadline of τ_i, as shown in Figure 4.2a.

But τ_x has a longer busy period, illustrated by Figure 4.2b, in which it suffers both preemption by τ_j and τ_k because its busy period is larger than S_i whereas releasing τ_k with τ_x, τ_x will not suffer interference from τ_j since its busy period ends before $T_i - S_i$.

This shows that the offset transformation is incorrect that it may be over optimistic. To make things even worse, in calculating the *exact* worst-case response time of a task, it is necessary to examine all the possible combinations of higher priority task offset alignments as suggested in [117] by Tindell, which has been shown to be intractable in general. In particular, the critical instant can occur at any of the combinatorial offset alignments of higher priority prologue-epilogue pairs.

In the case of our task model where there are only two tasks (prologue and epilogue) in an offset relation, for each task there are $O(2^n)$ number of the above response time tests to perform, where n

is the number of higher priority prologue-epilogue pairs (base two because there are two tasks in a single offset relation) and that a single test requires a pseudo-polynomial time calculation [8]. The time complexity of the whole procedure is therefore expotential with regard to the number of imprecise tasks.

Note that because calculating the exact response time for a task set with sizeable number of imprecise tasks can be very computationally expensive, equation (4.2), albeit pessimistic, can always be used as a *sufficient* test for verifying schedulability (still with intermediate deadline assignment but without offsets for tasks).

4.2.5 A Simplified Test Under DM Priority Ordering

Below it is shown that, in calculating the worst-case response time where the partner task is of higher priority, equation (4.5) alone actually accounts for the worst-case response time of τ_j by proving the following theorem. That is, the value of equation (4.6) can not exceed that of equation (4.5) if schedulability is assumed, where DM ordering and the intermediate deadline assignment (Section 4.2.1) are applied.

Theorem 1. *If task τ_j is the prologue or epilogue of an imprecise task τ_i where τ_k is the partner task with higher priority than τ_j, and their intermediate deadlines and offsets are assigned according to Section 4.2.1 with priorities in DM ordering, the worst-case response time of task τ_j is the first busy period ignoring any possible interference from τ_k.*

Proof. The idea is to show that the second busy period can not be greater than the first busy period if the task τ_j is to remain schedulable since the first busy period will be greater than D_j if that is the case.

Since τ_j and τ_k are in the same imprecise task, according the rules of assigning intermediate deadline (the shorter the WCET the shorter the deadline), for τ_k to have a higher priority based on DM, $C_k \leq C_j$. Comparing the equations for calculating the two busy periods, if one ignores the first term $\left\lceil \frac{w_j^n - O_j}{T_j} \right\rceil C_j$ of equation (4.6),[4] the value from equation (4.6) must be less than or equal to that of equation (4.5).

Now the condition where the first term will take effect in the equation is examined. Observe that the ceiling function can only have value larger than zero when $w_j^n - O_j > 0$ since tasks' deadlines and subsequently their offsets must be less than period from the assumptions. The resulting value of the first

[4]Note that the result of the ceiling function of the term could not be negative because $O_j \leq D_i \leq T_i$.

term has the largest value as C_j.

The only circumstance where $w_j^n - O_j > 0$ is when the busy period execute up to the offset O_j, which is also the deadline of τ_k. For any values of C_j and C_k, the window that is allowed for higher priority task's preemption in both tasks, which is $D_j - C_j$ and $D_k - C_k$ respectively, are equal. This fact follows directly from the intermediate deadline assignment scheme defined in Section 4.2.1.

Therefore, if the busy period of equation (4.6) exceeds O_j, that is C_k plus all the interference of τ_j is greather than $O_j = D_k$, the busy period of equation (4.5) would be greater than D_j. In fact for both of their busy periods to be equal, $C_k = C_j$ so that the two tasks are having adjacent priorities, and thus suffering from the same interference.

□

It can be concluded that equation (4.5) accounts for the worst-case response time of any task in the system (equation (4.4) also reduces to equation (4.5) because it has just been proved that a partner task of higher priority can not interfere another's schedulability), resulting a simpler test than the previous one proposed in [7], which is in turn based on Tindell's analysis of arbitrary offset [117]. The main improvement comes from how prologue task and epilogue task are interrelated and the way their deadlines and offsets are assigned, rendering any deemed interference from the partner task impossible. Please note that the simplified test does not apply in the enhanced version of scheduling described in the next chapter. Both the busy periods will need to be examined for each offset combination.

Schedulability of the task system can be easily verified by comparing the response time to the deadline for all tasks. That is:

$$\forall j \in \gamma : r_j \leq D_j, \tag{4.7}$$

where γ is the set of all non-imprecise tasks plus all prologue and epilogue tasks in the task system.

If the blocking factor B_i is omitted, and the concerned system only consists of sporadic tasks, the test is exact because the worst-case can indeed occur. However, in the case of periodic tasks with shared resources bounded by a blocking factor, the worst-case interference may never occur due to particular phasing caused by how tasks access resources, making the test sufficient but not necessary.

4.3 Summary

In this chapter new results in scheduling imprecise computation based on fixed priority preemptive scheduling have been presented. The previous analysis in scheduling imprecise computation of such task model [7] is not sufficient in terms of the offset transformation in calculating task's response time, for which proposed fixes are provided here. The general method for determining the exact worst-case response time with offset is exponential in time with the number of imprecise tasks concerned. The tests were further simplified by proving that under DM priority ordering and the deadline assignment scheme, the previously proposed test for second busy period is not necessary.

In the next chapter a novel scheduling scheme, based on Dual Priority Scheduling, to maximize the scheduling of unbounded optional components in the system, along with the required schedulability tests, is presented.

Chapter 5

iDPS - imprecise Dual Priority Scheduling

In the previous chapters the P-O-E task model for imprecise computation was introduced along with the exact schedulability test.

This chapter presents the details of a new scheduling scheme, based on Dual Priority Scheduling, which shows how the scheduling of optional components can be enhanced while still maintaining system schedulability. Simulation studies were performed and the results are shown for performance assessment. Scheduling for aperiodic tasks within the proposed scheduling framework is also discussed.

5.1 iDPS: imprecise Dual Priority Scheduling

In scheduling imprecise computation with the task model concerned, the main objective is to try servicing as much unbounded optional computation as possible while providing hard guarantees for mandatory tasks. However, as explained in the previous chapter, optimally scheduling optional tasks among hard tasks with offsets, in general, is NP-hard [76, 7]. Nonetheless, based on the response time analysis formulated above, an effective scheduling scheme can be derived with satisfying performance by utilizing the concepts from Dual Priority Scheduling (DPS) [43] while maintain the required hard timing guarantees.

DPS, proposed by Davis and Wellings, is characterized by its low overheads and simple implementation. It has been shown highly efficient for servicing aperiodic tasks. To enable DPS to service imprecise tasks with the proposed task model, the way it sets the promotion time of different tasks needs to be modified. The term *iDPS* is used to refer to the extended version of DPS which is aimed at supporting

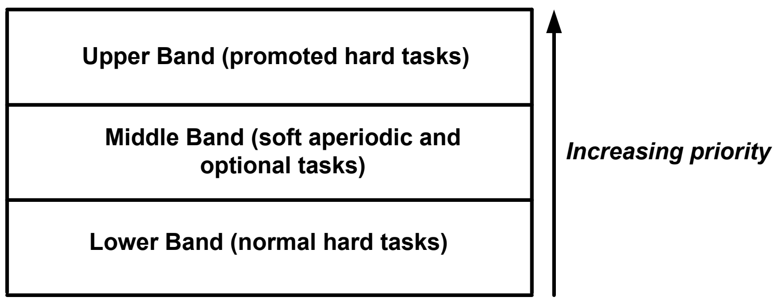
Upper Band (promoted hard tasks)
Middle Band (soft aperiodic and optional tasks)
Lower Band (normal hard tasks)
Increasing priority

Figure 5.1: Priority hierarchy of DPS.

imprecise computation.

In DPS, the worst-case execution time of a task is first found by the response time equations, such as equations 4.5 and refeq4.6 in this case. Note that these equations can easily be extended to accommodate release jitter and arbitrary deadlines as shown in [43].

There are three priority bands in DPS (Figure 5.1), aperiodic tasks reside in the middle band and any tasks in a higher band have absolute higher priorities than the ones in a lower band. Hard tasks, when first released, are set to run in the lower band. They are promoted to the upper band when it is necessary for them to meet their deadlines. The latest promotion time $y_i = t_i + Y_i$, where t_i is the original release time and $Y_i = (D_i - R_i)$ is found by first calculating the worst-case response time R_i of the task τ_i. In other words, if τ_i has not been executing after an interval Y_i since its release, it will be promoted at the instant y_i.

The original DPS sets *every* hard task's promotion time as late as possible for responsiveness of aperiodic tasks. In iDPS, the same promotion strategy for non-imprecise hard tasks is retained. However, to maximize the available time for scheduling optional tasks, it is necessary to schedule the prologue task as early as possible and the epilogue task as late a time as possible, so that the chance of scheduling pending optional tasks is maximized.

More specifically, in iDPS the promotion strategy for tasks is modified so that for each prologue task τ_i^p the promotion time is set to their initial release time t_i, which is the beginning of every period of the task. For each epilogue task τ_i^p and any normal hard tasks τ_x, a latest promotion time y_i is set to $t_i + (D_i - R_i)$, as shown in Figure 5.2 (prologue and epilogue represented by C_j and C_k). In such a way, the interval between prologue and epilogue is effectively enlarged.

Note that the accuracy of WCET, which are assumed to be given, is important - as the epilogue task is postponed as much as possible, invalid higher WCET may lead to an overrun of tasks and subsequently

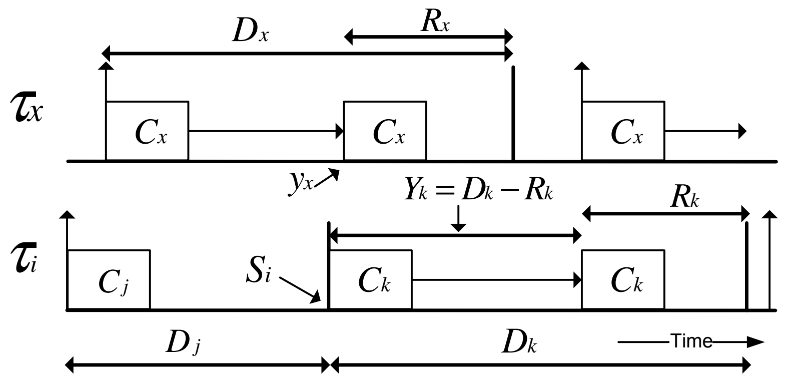

Figure 5.2: Delayed promotion time of epilogue task and ordinary task by iDPS.

missed deadlines. Note, however, that this is rather a problem of inaccurate WCET analysis than a problem of iDPS, and a safety margin can easily be incorporated if it is deemed necessary.

5.1.1 Schedulability Analysis

According to Bernat and Burns [18], promoting a task at time y_i is equal to assigning the effective release time of the task the time $t_i + Y_i$ and a deadline $D_i - Y_i$, where t_i is its original release time. Note that in applying iDPS, unfortunately, when a task's offset is changed, the worst-case response time calculated according to the previous chapter will be no longer valid. In other words, if tasks are promoted at an offset according to the previous response time value calculated, the actual response time may not be the same since other tasks may be released at a different time in their execution, effectively changing the offset relation. In particular if the resulting response time is longer than the previous calculated response time, releasing a task at that offset will result in tasks missing their deadlines.

Therefore, to apply iDPS, the offset O_i of epilogue τ_i will need to be re-adjusted as $S_i + Y_i$, and the schedulability tests need to be re-run again for guaranteeing schedulability of the system.

As a solution, this offset re-adjustment and response time re-calculation iteration procedure is performed from the highest priority to the lowest priority epilogue task where the offset of the epilogue is its response time obtained in the last test, until all epilogue task's response time value have been fixed with all tasks schedulable. In case any lower priority task becomes unschedulable with the new offset value of an epilogue task used in an iteration, a fall back policy is employed where the values used in the last iteration will be adopted and no new promotion times will be calculated for that task. The iteration continues with the rest of the lower priority epilogue tasks.

Each iteration involves the schedulability tests mentioned in the previous chapter and the maximum number of iterations is bounded by the number of epilogue tasks in the system. Therefore the major complexity is still imposed by the number of imprecise tasks in the system as in the schedulability tests with offsets.

Schedulability of all tasks is now verified by comparing the response time to the deadline as:

$$\forall i \in \gamma : R_i \leq D_i - O_i, \tag{5.1}$$

where $O_i = 0$ if τ_i is a prologue task and γ is the set of all tasks in the system.

5.1.2 Effects of iDPS

Now the principle effects that can be expected from applying iDPS are presented. The actual amount of time that can be used for executing optional component of an imprecise task is a dynamic property of the system since it is a function of when the prologue finishes execution and when the epilogue is released. This can be made more quantitative by the following definition considering an imprecise task τ_i with prologue τ_j and epilogue τ_k:

Definition 1. *The interval, W_i, that can be used for executing the optional component of τ_i is the duration between the completion of its prologue task and the release time of its epilogue task where the prologue executes up to its worst-case response time.*

The exact increase for W_i by iDPS on τ_i depends on all the attributes (deadlines, response times, etc.) of different tasks in the system. Nevertheless, such an increase could be substantial - in fact it could be as much as or even greater than the original interval. In particular, it is shown that W_i is exactly doubled in a special case by the following theorem if $C_j = C_k$.

Lemma 1. *If the intermediate deadline is assigned according to Section 4.2.1, the new deadlines of τ_j and τ_k will be set as half of the original deadline $\frac{D_i}{2}$.*

Proof. This directly follows from equation (4.3) when $C_j = C_k$. □

Theorem 2. *If $C_j = C_k$ and the followings concerning τ_i hold:*

- *the prologue task executes up to its WCET;*
- *the prologue task experiences worst-case interference from higher priority tasks and;*
- *both prologue and epilogue do not access shared resources.*[1]

By applying iDPS, W_i is exactly doubled, that is, increased by 100% compared to the original offset defined in Chapter 4.

Proof. By lemma 1 it is known that prologue and epilogue have the same deadline, and by DM ordering their priorities will be adjacent to each other.[2] According to equation (4.5, 4.6), both of their worst-case response times will be the same.

The original W_i is the deadline of prologue minus its worst-case response time, i.e., $D_j - R_j$, since the epilogue task is released immediately at S_i which stops the execution of the optional task. By using iDPS, the release time of the epilogue task τ_k is postponed by an extra duration $D_k - R_k = D_j - R_j$. $\square$

At first glance the theorem seems rather restrictive since worst-case rarely happens and that prologue and epilogue can be expected to have different WCET values. It is provided here as a proof of concept showing the amount of improvement that can be obtained by exploiting response time information of tasks in iDPS. In fact, systems do show a similar performance gain in the simulations studied shown in the next section (when tasks have non equal execution times and are not experiencing worst-case interference).

In addition, because priority ordering is based on deadlines and that the deadlines of prologue and epilogue usually do not differ much compared to other tasks in the system (assuming the WCET to deadline ratio is small, equation (4.3)), their priorities will often stay close, which roughly resembles the required assumptions (adjacent priorities) and expected gain of the interval length.[3]

[1]The proof is also valid when both of them share the same resource bounded by the same blocking factor, and the prologue experiences the worst-case scenario.

[2]In cases where other tasks also have the same deadlines, one can always arrange their priority in such a way that they are adjacent to each other without compromising schedulability.

[3]Especially for high priority tasks with short deadlines since they suffer less interference.

Note that any unused guaranteed capacity (slack) will be automatically reclaimed and used in the middle band of the scheduling framework for enhancing system utility. Note also that the run-time overhead of iDPS should not be more than that of DPS (proved effective in [43]) since no sophisticated calculations and mechanism changes are involved.

5.2 Performance Evaluation

This section evaluates how effective the proposed task model and scheduling scheme can solve the problem of maximizing resource usage. Based on the timing requirements used in the case study presented in Chapter 1, a set of simulations were performed to model the RoboCup scenario.

There are three versions of the scenario which are of interest:

1. where the 11 agent tasks are represented by normal hard tasks (marked “Traditional” in the figures);
2. where the 11 agent tasks are represented by the P-O-E task model with schedulability verified as suggested in Chapter 4. Optional components are scheduled whenever no hard task is executing (marked “P-O-E” in the figures); and
3. where the 11 agent tasks are represented by the P-O-E task model plus that iDPS, as described in the last section, is applied so that normal hard tasks and epilogue tasks are scheduled as late as possible for facilitating optional computation (marked “iDPS” in the figures).

The 11 agent tasks are denoted by the set τ_a; the deadline of each task is equal to the period and assigned as 10ms. The total utilization of all agent tasks is denoted as $U_a = \sum_{\forall i \in \tau_a} \frac{C_i}{T_i}$. In addition, there are 10 more normal tasks with randomly generated values of period. The period values span across a few orders of magnitude (4 - 7 digit figures) in a roughly uniform way where the value of each digit is chosen randomly (1 - 9 for the most significant digit and 0 - 9 for the rest). This set of tasks mimic the workload of background system tasks in the case study, denoted as τ_s, where $U_s = \sum_{\forall i \in \tau_s} \frac{C_i}{T_i}$ represents the total utilization of these tasks. Note that the resolution of a simulation tick is one micro-second (μs), thus the tasks generated represent a reasonable period distribution of a real system.

Note that due to the sizable number of imprecise tasks in the simulations, all response time calculation is based on the tractable, sufficient (but not exact) test, as defined in Chapter 6. In the chapter it is shown that the difference between the response time of tasks calculated by the two tests is only minimal while

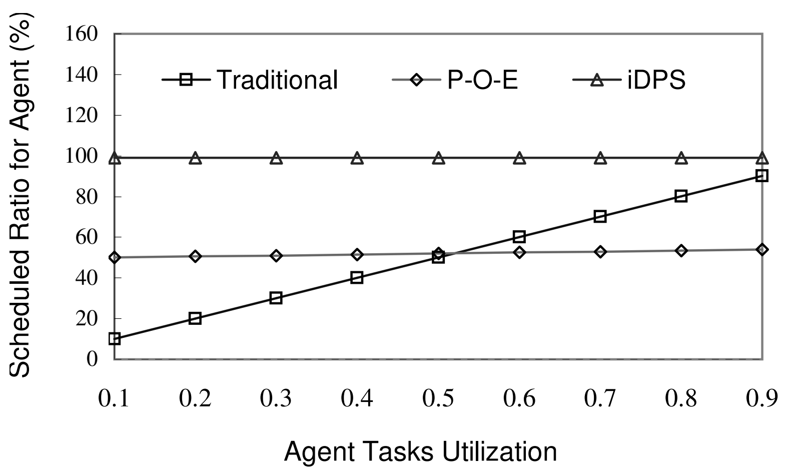

Figure 5.3: Performance of different scheduling schemes against varying agent tasks utilization.

the tractable is much faster to calculate. In addition, all mandatory tasks are assumed to not hold any resources (no blocking). Optional tasks are set to have the same utility so that they are scheduled in a round robin fashion (when eligible for execution) in the middle band throughout the experiments.

Except for the varying parameter in each experiment, other parameters are held at default values as follows:

- total utilization of system tasks $U_s = 0.01$ to allow more resources for agent tasks;
- total utilization of agent tasks $U_a = 0.8$;
- the unused WCET ratio $E = 1 - \frac{e}{C}$ is set to 0, where e is the actual execution time such that all tasks executed up to their WCET.

Moreover, to evaluate the flexibility of the P-O-E model, the imprecise version of U_a is only set at 20% of the traditional version of U_a. That is, the WCET guaranteed by the system is only 20% of the WCET value guaranteed for agent tasks without using the P-O-E model, where prologue and epilogue each shares 10%. Each simulation runs for 10,000,000 ticks and the primary performance metric is the ratio of the total time used for scheduling agent tasks (including all mandatory and optional computation) against the total time simulated.

Figure 5.3 shows this ratio against varying value of total utilization for agents (U_a). Each data point in the figure represents the averaged value calculated over 10 simulations each using a different set of randomly generated system tasks (τ_s) with the same set of parameters.

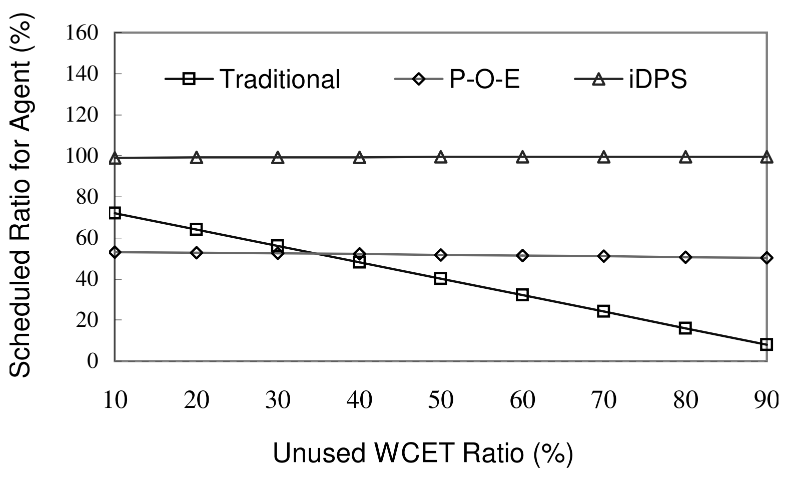

Figure 5.4: Performance of different scheduling schemes against varying unused WCET ratio.

It can be observed that as U_a increases the total time scheduled for agent tasks (Traditional) also increases accordingly. This must be true since more capacity is guaranteed for the agent tasks assuming the whole system is schedulable. The performance of scheduling with the imprecise task model (P-O-E) and further with iDPS are interesting. They remain steady at about 52% and 99% respectively. This means that even when their WCET are set at 20% of the agent task utilization without P-O-E, the amount of optional components scheduled allows them to make use of the spare capacity from the system.

In particular, since the periods of the imprecise agent tasks are in sync, employing the P-O-E task model makes all epilogue tasks having the same offset S (Section 4.2.1) relative to the prologue tasks. Therefore the interval between S to the end of the period is not available for scheduling optional tasks. With increasing value of U_a the performance of "P-O-E" eventually falls below that of the traditional method. However, scheduling by iDPS remains the best approach of the three. Based on the worst-case response time information the promotion time of epilogue tasks are set as late as possible. This leads to almost all of the available capacity assigned to the imprecise tasks representing the 11 agent programs.

Nevertheless, it was assumed that the unused WCET ratio to be 0. This is unrealistic for real systems - especially for AI algorithms which may have even more variable and longer execution times than typical algorithms do. Figure 5.4 shows how the performance of the traditional approach changes with varying unused WCET ratio that is applied to all tasks in the system, while holding $U_a = 0.8$. It can be seen that as the value of E increases, the performance of the traditional approach drops accordingly.

Consider how difficult it is to construct AI algorithms at design time so that they can have very predictable execution times. Besides this issue, even when the system is fully utilized, it may lead to the

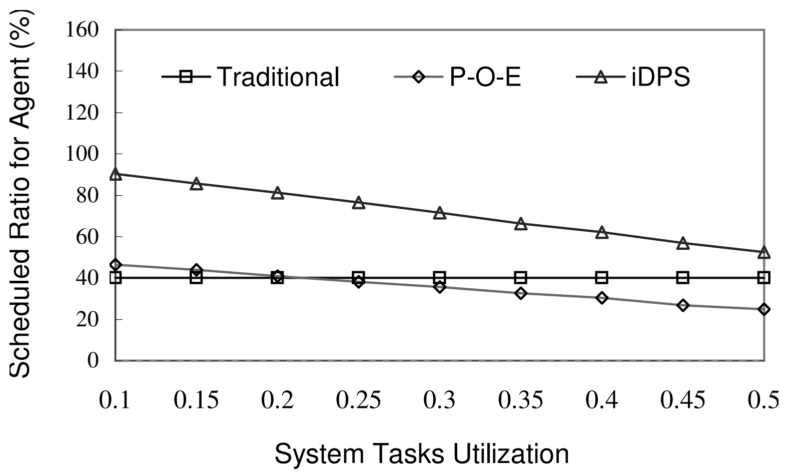

Figure 5.5: Performance of different scheduling schemes against varying system tasks utilization.

system being unable to admit new tasks online for an open real-time system. In contrast, the task model allows computation of different natures to be separated with much smaller WCET so that new tasks can be admitted to the system dynamically while the dual priority mechanism in iDPS makes sure any slack time from unused guaranteed time of hard tasks is reclaimed, and thus maintaining the superior performance.

This ability is also evidenced in Figure 5.5 which shows how the performance of iDPS varies as the system is increasingly utilized by system tasks, where U_a is held at 40% of the overall system workload. It can be seen that iDPS adapts to the changes in available resource dynamically and again outperforms the other approaches.

Moreover, there is yet another important advantage of using iDPS. That is, imprecise computations such as anytime algorithms tend to give very good results in a very short time but improve result quality slowly as time progresses [44, 124, 125]. By delaying the execution of epilogue task and non-imprecise tasks more optional components will get executed earlier, thus providing the desired result earlier, which could be important in, for example, real-time agent systems that have to operate under dynamic environments.

5.3 Scheduling Aperiodic Tasks

As mentioned, DPS was originally proposed as a way to efficiently schedule aperiodic tasks [43]. If aperiodic tasks are soft without need for hard guarantees, they may be added to the middle band where

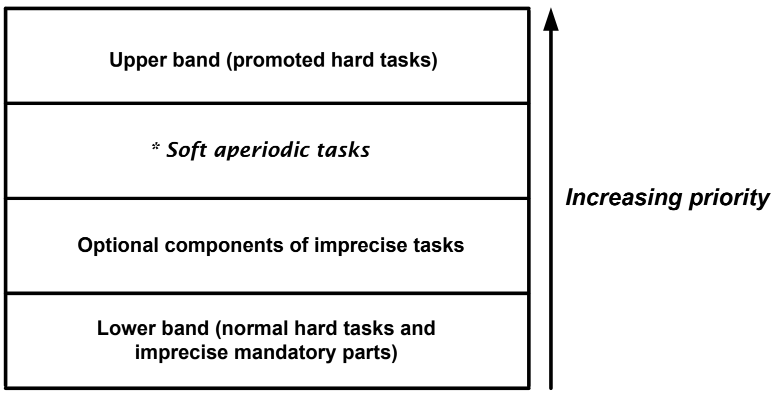
Upper band (promoted hard tasks)
* Soft aperiodic tasks
Optional components of imprecise tasks
Lower band (normal hard tasks and imprecise mandatory parts)
Increasing priority

Figure 5.6: Extending iDPS for aperiodic tasks scheduling.

optional tasks reside. Note also that iDPS is a flexible framework where many levels can exist. In systems where the aperiodic workload has higher utility than other optional tasks, or average responsiveness of aperiodic tasks is important, one can introduce an extra priority band into the scheduling hierarchy in such a way that any ready aperiodic tasks will preempt all the optional task executions. Figure 5.6 shows an example priority hierarchy.

Some aperiodic tasks may be firm tasks, requiring to be completed before some deadline or their utility becomes zero. However, Davis and Wellings [43] showed that the time complexity of exactly calculating the available slack in DPS, up to an arbitrary time instant, is pseudo-polynomial. Calculations depend on task periods and deadlines of hard tasks and may prove to be impractical when online acceptance of aperiodic tasks is required.

An alternate method is to make use of a periodic server, as described in Chapter 3, within the iDPS framework. Previous analysis of periodic servers depends on utilization test, which often assumes the server operates at high priority only for deriving the analytical schedulability test [73, 113]. In serving aperiodic tasks by periodic servers, however, Bernat and Burns [20] found that a large server capacity is actually the main contributing factor for task's responsiveness. For example, a periodic server can have a long period and low priority but with a relatively large capacity. In particular, they showed that a Deferrable Server can be modelled as a sporadic task with release jitter, which can be analysed using the response time analysis. Hence it could be incorporated into iDPS.

The use of multiple servers [20] may also be of use for firm aperiodic tasks of very different timing properties. Single periodic servers assumes a simple queue scheme and therefore may cause urgent aperiodic tasks to miss deadlines. If the load of firm aperiodic tasks is known a priori, extra servers can be used to match the timing requirements with designated server parameters (period, capacity, etc.).

5.4 Summary

In this chapter new results in scheduling imprecise computation based on fixed priority preemptive scheduling were presented for improving the scheduling of optional components based on an extended version of DPS [43], called iDPS. The improvement can be substantial as illustrated by the simulations conducted. Suggestions on how aperiodic tasks can be scheduled in the framework were also provided.

In the next chapter a more tractable, sufficient but not exact, version of the schedulability test is presented.

Chapter 6

A Tractable Schedulability Test for Imprecise Computation

In Chapters 4 and 5, schedulability test for imprecise computation of "Prologue-Optional-Epilogue" task structure based on fixed priority real-time scheduling was developed where a particular scheduling scheme, named "imprecise Dual Priority Scheduling" (iDPS), was shown to perform significantly better than a previous approach as shown in the previous chapter [7].

The time complexity of the schedulability tests developed previously, however, are exponential with regard to the number of imprecise tasks in the system. The tests involve the testing of all the possible combinatorial combinations of offset alignments (released at the same time) of different offset transactions. For each offset alignment combination, a whole schedulability test needs to be performed and the maximum value of all resulting busy periods is taken as the worst-case response time. Unfortunately, the tests become computationally prohibitive with even a moderate number of imprecise tasks.

Tindell devised a tractable offset test for tasks with any offset relations using a different approach [117]. In essence, the idea is that, rather than performing a whole test for each offset combination, one can examine the interference caused by using all different offset relations and take the maximum of them *during* each iteration of the calculation of the worst-case response time. Employing this approach a tractable schedulability test, which is a special case of Tindell's analysis, is developed. The time complexity of calculating the busy period for each task is reduced from the original $O(2^n)$ to $O(2n)$, where n is the number of higher priority imprecise tasks in the system - a critical difference as n increases.

In this chapter the exact schedulability test required for the task model concerned, described in the two previous chapters, is re-visited first. The details of the tractable test derivation are then presented, followed by a set of extensive simulations for performance evaluation. The simulations are conducted with a wide range of different parameters, including the number of imprecise and non-imprecise tasks in the system, system utilization and utilization skew of tasks. The results are discussed in detail before the concluding remarks.

6.1 Exact Schedulability Test

In the original schedulability analysis by Audsley et al. [11, 7] for the P-O-E task model, two busy period intervals need to be examined to determine the worst-case response time, since a partner task of a task τ_i with higher priority can cause other tasks with intermediate priorities to finish later, effectively causing interference to τ_i itself. It was shown in Chapter 4 that the second busy period is actually not required for verifying schedulability when tasks are in the deadline monotonic (DM) order where the offset of each epilogue C_i^p equals the intermediate deadline S_i of the imprecise task, defined as:

$$S_i = \frac{D_i - (C_i^p + C_i^e)}{2} + C_i^p. \tag{6.1}$$

Treating every prologue and epilogue as normal tasks, the number of tasks during schedulability test is now $2n + m$, where m is the number of normal tasks while n is the number of imprecise tasks in the system. Indexed by j, the busy period of every task τ_j in the system can be calculated as:

$$w_j^{n+1} = C_j + B_j + \sum_{x \in \mathbf{hp}(j) \wedge x \neq k} \left\lceil \frac{w_j^n - O_x}{T_x} \right\rceil C_x, \tag{6.2}$$

where τ_k is the partner task of τ_j, which belongs to an imprecise task.

However, to enlarge the available interval between prologue and epilogue during which more optional components can be scheduled, Dual Priority Scheduling [43] has been applied to normal tasks and epilogue tasks as described in the previous chapter. Since the offset of epilogue task is changed (not in DM ordering) due to the effort of promoting epilogue tasks as late as possible, the second busy period equation, stated below, is needed for the re-evaluation of task's schedulability:

$$w_j^{n+1} = \left\lceil \frac{w_j^n - O_j}{T_j} \right\rceil C_j + B_j + C_k + \sum_{x \in \mathbf{hp}(j) \wedge x \neq k} \left\lceil \frac{w_j^n - O_x}{T_x} \right\rceil C_x. \tag{6.3}$$

The actual busy period is $w_j - O_j$ after the value w_j converged, since task τ_j is assumed to be released with an offset O_j relative to its partner task. At the end the first and second busy periods are compared and the larger value is used as the actual worst-case response time.

However, to account for the *exact* worst-case response time of a task, it is necessary to examine all the possible combinations of higher priority task offset alignments, which has been shown intractable in general [117]. In particular, the critical instant can occur at any of the combinatorial offset alignments of higher priority prologue-epilogue pairs.

In the case of our task model where there are two tasks (prologue and epilogue) in an offset relation, for each task there will be $O(2^n)$ number of the above response time test to perform, where n is the number of higher priority prologue-epilogue pairs (base two because there are two tasks in a single offset relation) and that a single test requires a pseudo-polynomial time calculation [8] in the worst-case.

6.2 Tractable Offset Test

Tindell devised a tractable offset test for tasks with any offset relations using a different approach [117]. In essence, the idea is that, rather than performing a whole test for each offset combination, One can examine the interference caused by using different offsets within a given transaction[1] and take the maximum of them *during* each iteration of the calculation of the busy period w. The time complexity of this procedure is reduced from $O(m^n)$ to $O(nm)$, where m is the number of tasks in transactions and n is the number of transactions.

In the P-O-E task model there are only two tasks in one transaction - the prologue and epilogue tasks, where the offset is assigned by defining an intermediate deadline between the two tasks. One can apply the analysis of Tindell to our task model and define it more formally as below.

Ignoring the blocking term, equation (6.2) can be rewritten as:

$$w_j^{n+1} = C_j + \sum_{t \in \mathbf{trans} - \mathbf{trans}(j)} \max_{x \in \mathbf{hp}(j) \cap \mathbf{tasks}(t)} (I_t(x)) \tag{6.4}$$

where

$$I_t(x) = \sum_{y \in \mathbf{hp}(j) \cap \mathbf{tasks}(t)} \left\lceil \frac{w_j^n - O'_y}{T_y} \right\rceil C_y. \tag{6.5}$$

$trans$ refers to the set of all transactions (all imprecise tasks) and $trans(j)$ returns the particular transaction where task τ_j is in, whereas $tasks(t)$ represents all the tasks in the transaction t. In equation

[1]A transaction consists if a set of tasks sharing the same period and having an offset relation to the beginning of it.

(6.5) O'_y refers to the relative offset of task τ_y to the release of task τ_x, therefore in equation (6.4) each offset combination in the same transaction will be tested and the maximum value will be taken, during each iterative calculation of w.

The second busy period equation is needed when iDPS is applied for the re-evaluation of the system schedulability when the offset of epilogue task is changed. The equation can be obtained in a similar fashion:

$$w_j^{n+1} = \left\lceil \frac{w_j^n - O_j}{T_j} \right\rceil C_j + C_k + \sum_{t \in \mathbf{trans} - \mathbf{trans}(j)} \max_{x \in \mathbf{hp}(j) \cap \mathbf{tasks}(t)} (I_t(x)) \tag{6.6}$$

where

$$I_t(x) = \sum_{y \in \mathbf{hp}(j) \cap \mathbf{tasks}(t)} \left\lceil \frac{w_j^n - O'_y}{T_y} \right\rceil C_y. \tag{6.7}$$

The time complexity of the test for each task is reduced from the original $O(2^n)$ to $O(2n)$, where n is the number of higher priority prologue-epilogue pairs (transactions) of the task. Note the time complexity for verifying the schedulability of the whole system is still pseudo-polynomial [8] with regard to the number of tasks in the system.

6.3 Performance Evaluation

In the simulations conducted by Tindell [117], with large numbers of randomly generated task sets, the response time of the lowest priority task is calculated by the exact test is compared to that from the tractable test. It was observed that the tractable test performed very close to the exact test.

Since our task model is a special case in the sense that there are only 2 tasks in every transaction with offset relations, it will be interesting to compare the performance of the tractable test derived here to the previous result. However, in addition to system utilization, one would like to understand the performance of the test under a wider range of task parameters including varying numbers of imprecise tasks, numbers of normal tasks and skewness of utilization distribution among tasks.

In all the simulations, DM priority ordering is assumed. Given the utilization of the system u, a task τ_i (including all prologue, epilogue and normal tasks) is randomly generated with period values spanning across a few orders of magnitude (4 - 7 digit figures) in a roughly uniform way where the value of each digit is chosen randomly (1 - 9 for the most significant digit and 0 - 9 for the rest). Deadlines are chosen to be at about 90% of the period value with small variations such that $C_i \leq D_i \leq T_i$..

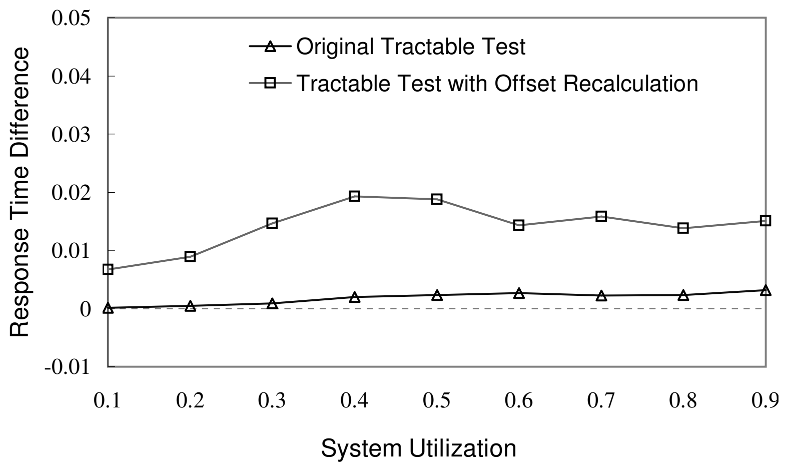

Figure 6.1: Tractable test performance against varying system utilization.

Note that the resolution of a simulation tick is one micro-second (μs), thus the tasks generated represent a reasonable period distribution of a real system. All mandatory tasks are executed up to their WCET and do not hold any resources (no blocking). Once a period is generated, the WCET C of the task is then calculated to provide the desired system utilization. Further, one of the normal tasks will be generated with the skew parameter s as:

$$C = s * u, \tag{6.8}$$

where the WCET of remaining tasks are defined as:

$$C = (1 - s) * u. \tag{6.9}$$

Except the varying parameter in each experiment, all the other parameters were held at default values as follows:

- system utilization $u = 0.5$;
- number of imprecise tasks $n = 5$;
- number of normal tasks $m = 20$;
- utilization skew $s = 1/(2n + m)$ such that all tasks equally share u;

The primary performance metric is the average task response time difference F of a task set, which is defined as:

$$F = \sum_{i=1}^{2n+m} \frac{R_i^t - R_i^e}{R_i^e}, \tag{6.10}$$

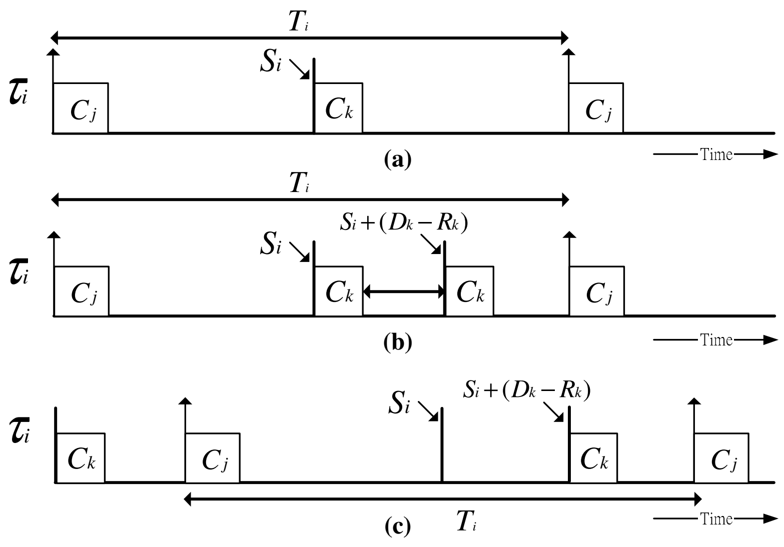

Figure 6.2: Explanation of greater response time difference in tractable test with offset recalculation.

where R_i^t is the worst-case response time of task τ_i calculated by the tractable test and R_i^e the one obtained by the exact test. Since there are two versions of F of interest, F_o denotes the average task response time difference obtained by the tractable test (equation 6.4), and, F_{dps} as the F from the tractable test with offset recalculation, obtained by the maximum value from equation (6.4) and (6.6).

Figure 6.1 shows the values of F_o and F_{dps} against varying system utilization. For statistical significance, each data point in the graph represents the average F over *100* randomly generated task sets.

It can be seen that the response time difference of the original tractable test, F_o, is negligibly small and very steady, which is about 0.0017% on average. The other line in the figure shows F_{dps} with varying system utilization. Although it seems relatively higher than F_o, note that the actual value is still very small (averaged 0.014%). The difference between the two versions can be explained by the fact that, in the original version without offset recalculation, the offset of an imprecise task τ_i was set according to the intermediate deadline S_i by equation (4.3), which evenly divides the available slack for scheduling prologue and epilogue (shown respectively by τ_j and τ_k in Figure 6.2a).

However, with the offset recalculation the effective offset of the epilogue is changed to a new value, $S_i + (D_k - R_k)$, as shown in Figure 6.2b. With this new offset relation, the distance between prologue and epilogue is shortened which means that a lower priority task can suffer a longer interference from the two tasks (Figure 6.2c). This will increase the chances where unnecessary interference can accumulate in the tractable test since in each iterative calculation the maximum value of the two busy periods resulted from the two different starting offset is taken.

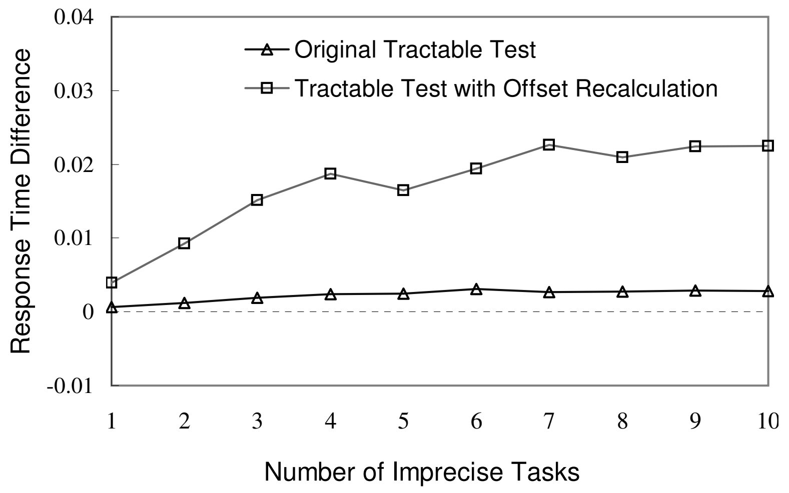

Figure 6.3: Tractable test performance against varying number of imprecise tasks.

The slight upward trend of both F_o and F_{dps} with increasing utilization can be understood by realizing that higher utilization implies the WCET to period ratio of all the tasks is higher. Therefore longer interference by tasks can prolong the busy period even more, hence the result.

Figure 6.3 shows the averaged F_o and F_{dps} over 100 randomly generated task sets with varying number of imprecise tasks. It can be observed that with increasing number of imprecise tasks the performance tends to degrade. This again can be explained similarly by the likelihood of unnecessary interference accumulation argument above, since with more imprecise tasks the number of iterations that need to be performed also increases. Note that because the exact test easily becomes intractable it has been difficult to compare the performance of the two tests with higher number of imprecise tasks.

Figure 6.4 shows the averaged F_o and F_{dps} over 100 randomly generated task sets with varying utilization skew level. The utilization skew seems to correlate well with response time difference for F_o while not showing a clear association with the performance of F_{dps}.

Two factors may both affect the performance here: 1) as more utilization is assigned to the skewed task, the WCET to period ratio of other tasks will decrease. Hence in general the interference of higher priority tasks on a particular task will be lowered also, thereby decreasing the likelihood of accumulating unnecessary interference in the tractable test; 2) on the other hand, as more utilization is assigned to the skewed task, if this happens to be a high priority task it will exert more interference on lower priority ones, thus lengthening the resulting busy period. Therefore the actual influence from skewness can not be predicted directly for F_{dps}, while the first factor appears dominant in the performance of F_o.

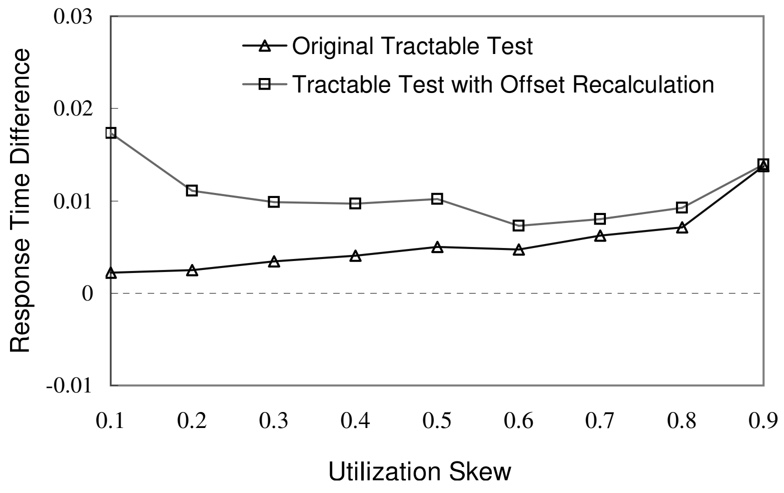

Figure 6.4: Tractable test performance against varying utilization skew.

It is interesting to see if the number of non-imprecise tasks in a system also makes a difference in performance. Figure 6.5 again shows the averaged F_o and F_{dps} over 100 randomly generated task sets, this time with varying number of normal tasks.

The lines appear to be decreasing with increasing number of normal tasks. This makes sense because the greatest contributing factor of the response time difference is how likely unnecessary interference can accumulate during the calculation of the response time equation, which is affected mainly by system utilization and the number of imprecise tasks as shown. Holding the difference ratio more or less the same, as more normal tasks are added to the system, the same ratio has to be divided by more tasks for obtaining the *averaged* response time difference, thus the smaller values depicted in Figure 6.5.

The simulations of the tractable test, when compared to that performed by Tindell, appear to perform better. One way to explain this is that there are only 2 tasks in an offset relation in each transaction where in Tindell's simulations 1-6 tasks are generated in 1-6 offset transactions randomly, presenting a more varied offset relations among tasks. As such, during the iterative calculation of the response time equation, it is easier for more unnecessary interference to take effect due to increased different combinations of task offset relations, therefore causing a longer busy period.

Note that the metric employed throughout the experiments provides a more general assessment of the test than the metric used in [117], which compares the response time difference produced by the tractable test for the lowest priority task in the system. One may think that the metric in use may introduce an averaging effect so that the tractable test for our task model seems to perform better. However, this is not true. It has been observed from the experimental data that the average response time difference

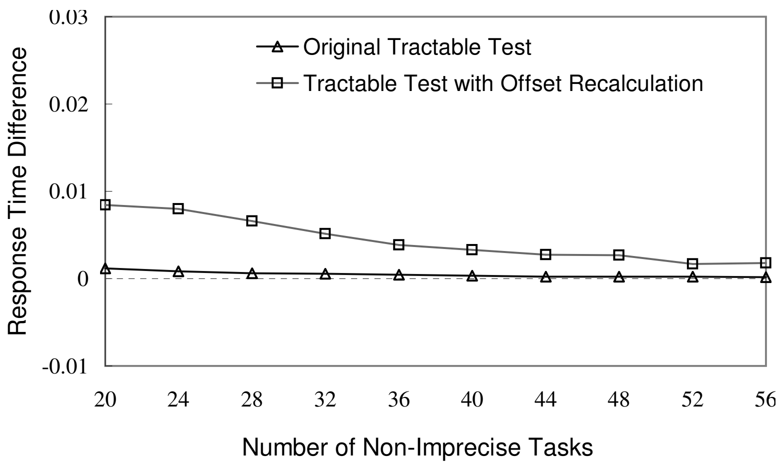

Figure 6.5: Tractable test performance against varying number of non-imprecise tasks.

of a whole task set can be greater than the response time difference of the lowest priority task in many occasions, meaning that it does not always suffer the worst unnecessary interference accumulation during the tractable test calculations for different offset relations.

Note also that, using the tractable test, the available interval for scheduling optional components of an imprecise task can be shortened due to the fact that the latest promotion time of an epilogue task is pushed earlier, caused by a more pessimistic worst-case response time calculated by the tractable test.

However, it is emphasized that the difference is extremely small - for example, even with iDPS offset recalculation the maximum observed value of F_{dps} in the 1000 randomly generated task sets in Figure 6.3 was only 0.089%. In addition, the tractable test may well be the only means for calculating response time when performing the exact response time test is computationally infeasible.

6.4 Summary

The exact real-time schedulability test for imprecise computation of the “Prologue-Optional-Epilogue” structure is computationally prohibitive when there are a large number of imprecise tasks present in the system. This motivates a more practical approach for verifying system schedulability. Based on Tindell’s analysis, a tractable test for the P-O-E task model is derived. Simulations show that the test performs extremely well under a wide range of different conditions. It is therefore a suitable substitute when the exact test becomes intractable. The result provides a more practical approach for schedulability analysis for imprecise computation of the task model concerned. In the next chapter the problem of assigning priority for tasks for maximizing system utility is considered.

Chapter 7

Optimal Priority Ordering for Imprecise Computation - A Utility-Based Approach

In fixed priority preemptive scheduling, task priority is static and assumed to be unchanged throughout a system's lifespan. The assignment of task priority in a hard real-time system has only one objective, which is to guarantee that all the tasks in the system are schedulable with their assigned priorities, thereby meeting their respective deadlines. However, real-time AI applications require a more adaptive resource management in the system [44, 125]. This means that only satisfying system schedulability will be inadequate in supporting such applications.

For example, in the context of Dual Priority Scheduling for imprecise computation (iDPS) presented in Chapter 5 [83, 7], a higher priority prologue task will finish its execution earlier (suffering less possible interference from higher priority tasks) and a higher priority epilogue task will have a later promotion time, which leads to a larger interval in which an optional component can be scheduled. Therefore, it will be more reasonable to assign higher priority to imprecise tasks whose optional components have higher values to the system.

Although assigning priorities based on tasks' deadlines is usually a general rule for optimal scheduling, Audsley [4] showed that if arbitrary offsets are allowed, neither rate monotonic priority ordering (RMPO) nor deadline monotonic priority ordering (DMPO) is optimal. He devised an optimal priority ordering algorithm which integrates with a feasibility test in determining task priority. The algorithm is guaranteed to find a feasible ordering whenever one exists. Although it is applicable to our task model (allowing offsets), however, the algorithm only concerns schedulability while the maximization of system utility based on the importance of tasks has not been considered.

Aguilar-Soto and Bernat [2] presented an optimal priority assignment algorithm which finds a priority ordering for a task set maximizing a given quality of service (QoS) measure that can be associated with task's priority. The algorithm, unfortunately, assumes the optimality of DMPO in determining priorities and is associated with an inefficient schedulability testing procedure, which are both unsuitable for the task model concerned here.

These previous works therefore motivate the development of an efficient algorithm that can find optimal priority orderings based on a task's utility. The rest of this chapter is organized as follows: first the priority ordering algorithm of Audsley is revisited in the next section before looking at the algorithm by Aguilar-Soto and Bernat. The details of their algorithms and the reason why they are not suitable for our scheduling framework are discussed. A new utility-based priority ordering algorithm is then presented, with proofs of correctness and optimality. This is followed by a set of simulations examining the performance of the proposed algorithms. The use of a tractable schedulability test in the UBPO algorithm is also studied and evaluated before the work is summarised.

7.1 Audsley's Algorithm

The priority ordering algorithm devised by Audsley [4] addressed the problem of assigning priorities to tasks with arbitrary start time (offsets). When offset is allowed, neither RMPO nor DMPO is optimal in assigning task's priorities. That is, there could be other priority orderings which are feasible but that RMPO and DMPO are infeasible. In such situation, in the worst-case, one has to examine all the $N!$ priority orderings for a system with N tasks. Audsley's algorithm reduces such complexity by proving the following theorem:

Theorem 3. *Audsley [5]. If process p is assigned the lowest priority and is feasible, then, if a feasible priority ordering exists for the complete task set, an ordering exists with process p assigned the lowest priority.*

The algorithm assigns priority using a bottom-up approach from the lowest priority to the highest, since once a task has been chosen for the lowest priority level, the theorem still holds for the rest of the task set - the same procedure can therefore be applied iteratively to a smaller and smaller set of tasks. This priority ordering algorithm is optimal with respect to any scheduling test as long as task worst-case response time are monotonically non-increasing with higher task priority; hence it can also be used with the schedulability tests derived in previous chapters for imprecise computation of P-O-E task structure.

However, the original algorithm only assumes arbitrary ordering of tasks before running. That is, there is no attempt to associate higher priority with more important tasks. Observe that more than one feasible priority ordering may exist, to maximize the utility of scheduling optional components within the framework of iDPS, the objective is therefore to find a priority ordering which, in maintaining the schedulability of a system, maximizes the interval between the prologue and epilogue tasks.

In terms of fixed priority scheduling, it can be achieved by assigning higher priorities to imprecise tasks whose optional component has higher value than others, since the higher the priority the earlier prologue tasks can finish their execution and the later the promotion time can be set for epilogue tasks; the time interval for potential optional task execution is thus enlarged.

7.2 Aguilar-Soto and Bernat's Algorithm

Aguilar-Soto and Bernat [2] presented an optimal algorithm, named "D&I", which finds a priority ordering from a task set maximizing a QoS measure associated with task priority. In particular, the algorithm finds a schedulable priority ordering which has the minimum lexicographical distance from the most desired ordering, which is determined by a task importance function.

The algorithm employs a branch and bound mechanism, where priority ordering is modelled as a tree searching problem. From highest to lowest priority, at each iteration it tries to assign a priority to a task (an entry point leading to a subtree of possible orderings), where lower priorities tasks are ordered in the DMPO fashion. If any task becomes unschedulable that subtree is abandoned and another task is selected for the concerned priority.

Although it also takes $O((N^2+N)/2)$ steps to find the ordering like Audsley's algorithm, in each step where a priority level is tested, each lower priority task has to be verified to be schedulable. In contrast, each step of Audsley's algorithm only involves testing the response time of the task concerned.

In addition, the algorithm assumes that when testing a priority for a task, all the lower priority tasks are ordered according to DMPO. However, as mentioned DMPO is not optimal for tasks with offsets. In other words, if the lower priority tasks ordered in DMPO is not schedulable, there may be other orderings which can make the task set schedulable. This renders the branch and bound mechanism used in the algorithm non-optimal and backtracking is now required. That is, if M is the number of lower priority tasks whose priorities have not been determined in each iteration, in the worst-case it is necessary to examine all $M!$ extra orderings rather than the original M orderings, which is clearly inefficient.

7.3 Utility-Based Priority Ordering Algorithm

Inspired by the bottom-up approach of Audsley's algorithm, it is modified here for assigning priorities to tasks by taking into account task utility. It is shown that the UBPO algorithm (Figure 7.1) also minimizes lexicographical distance like the D&I algorithm of Aguilar-Soto and Bernat when a different measure is used. In particular, the algorithm tries to assign the lowest possible priority to a task with low important at any given time, where an optimal priority ordering sequence is one whose tasks are ordered from lowest priority to the highest priority, left to right, where lower priority is assigned to tasks with lower importance based on a given task preference function.

The definition for our version of lexicographical distance of priority ordering sequence is given below and similar notations are used as defined in [2]. In particular, a task system is denoted by $\tau = \{\tau_1, \tau_2, ..., \tau_N\}$, representing all tasks in the system, where N includes the total number of non-imprecise and imprecise tasks each represented by a pair of prologue and epilogue tasks.

If an order relation $\prec$ ("precede to") is defined over the task set, a sequence $S = \langle \tau_1 \tau_2 ... \tau_N \rangle$ can be obtained such that for any task position $j < k$ in the sequence $\tau_j \prec \tau_k$. The ordering relation is generic and any such relation that produces a total ordering over the task set can be employed as desired; in this context it may be referred to as a "less important than" relation. The set $\hat{S}$ represents the set of all possible $N!$ sequences for a system with N tasks. Further, S^D is denoted as the input sequence which is most desired by an order relation, and S^F referring to the final priority ordering sequence found by an algorithm.

The ordering defines priority preference also: if $\tau_j \prec \tau_k$, a lower priority is assigned to τ_j. For simplicity, priorities are assigned from left to right in a priority ordering sequence - clearly if such sequence is equivalent to the sequence defined by the ordering the most desirable priority ordering is obtained for all tasks.

However, since such a sequence may not be schedulable, the objective is then to find a sequence that is as close to the desired sequence as possible. Aguilar-Soto and Bernat [2] defined the comparison of

```
int Optimal_Priority_Ordering(Task[] TaskSet) {
    int Schedulable;
    for (int K=N; K>1; K--) {
        Schedulable = 0;
        for (int J=K-1; J>0; J--) {
            if (Schedulability_Test(TaskSet, K)) {
                Schedulable = 1;
                break;
            }
            Swap(TaskSet, K, J);
        }
        if (!Schedulable)
            return(UNSCHEDULABLE);
    }
    return(SCHEDULABLE);
}
```

Figure 7.1: The utility-based priority ordering (UBPO) algorithm.

any two sequences in $\hat{S}$ as follows: if $\alpha, \beta \in \hat{S}$, α is said to be more important than β if comparing each element $\alpha[i]$ with $\beta[i]$, where $i = 1...N$ starting from the leftmost to rightmost of two sequences, the first difference is in the k^th task and $\alpha[k] \prec \beta[k]$. The more important sequence is said to be lexicographically smaller. However, since the order relation used is "less important than" rather than "more important than", the *less* important sequence will be lexicographically smaller.

With the ordering relation, a function $I : \hat{S} \to [0, 1, ..., N! - 1]$ can be defined mapping all possible priority orderings to a unique number, where $I(S) = 0$ if S is the most desired sequence defined by the relation. The lexicographical distance of two sequences $\alpha, \beta \in \hat{S}$ can therefore be obtained by $I(\alpha) - I(\beta)$.

For example, let $\tau = \{a, b, c\}$ and the desired ordering $S^D = \langle bac \rangle$ (b the lowest importance), the lexicographical distance of the sequences are listed below:

$$I(\langle bac \rangle) = 0$$
$$I(\langle bca \rangle) = 1$$
$$I(\langle abc \rangle) = 2$$
$$I(\langle acb \rangle) = 3$$
$$I(\langle cba \rangle) = 4$$
$$I(\langle cab \rangle) = 5$$

Figure 7.1 shows the utility-based priority ordering (UBPO) algorithm, in which the original bottom-up approach by Audsley is adopted. Given a desired task set sequence ordered based on task's utility (a task importance ordering) as input, where high priority is assigned to task with higher utility, the algorithm finds a priority ordering that is both schedulable and lexicographically closest to the desired input ordering, if a feasible ordering does exist.

The outer loop in $Optimal\_Priority\_Ordering()$ serves to fix the next lowest priority K for a task (from left to right of a priority sequence as K decreases), where the inner loop iteratively swaps the next lowest priority task candidate to priority K and test its schedulability. In particular, if a task is not schedulable with the assigned priority, the inner loop swaps its position with the next candidate task indexed by J, until a schedulable task can be found.

After a task is found to be schedulable with an assigned priority, the iteration goes on to find the task for the next lowest priority until when $K = 1$ where there is only one task remaining and it is assigned the highest priority. Note that for simplicity purpose of the algorithm, without loss of generality, it is assumed that any task is schedulable when it is assigned the highest priority in the system so that there is no need to perform schedulability test on that task.

Theorem 4. *In fixed priority preemptive scheduling, the UBPO algorithm, using an exact feasibility test for determining task schedulability, is optimal in the sense that if there exists a feasible priority ordering it is guaranteed that the algorithm will find it.*

Proof. This optimality is inherited from Audsley's algorithm, based on the properties of fixed priority scheduling and response time test (Theorem 3) [4]. The original algorithm assumes the selection order of tasks in fixing priority can be arbitrary, the only difference between the two is that the UBPO algorithm makes use of a particular input ordering and task selection order.

Note that whether or not the algorithm can find a feasible ordering whenever one exists depends on the schedulability test employed. If a sufficient but not exact test is used, some orderings may be feasible but can not be found by the algorithm because it is a decision of the schedulability test. Since the schedulability test derived in Chapter 4 for imprecise computation is exact, the UBPO algorithm is also optimal in the sense that if there exists a priority ordering that makes a system schedulable it will find one. □

Theorem 5. *In fixed priority preemptive scheduling, the UBPO algorithm, using an exact feasibility test for determining task schedulability, is optimal in the sense that if a schedulable priority ordering is found, the lexicographical distance to the desired input ordering will be minimized among the set of all feasible orderings.*

Proof. It is first shown that, in each iteration, the swapping operation does not affect the importance ordering relation in the subsequence representing tasks whose priorities have not been determined. Then it is shown that fixing a priority of a task with such swapping method will produce a task sequence having a smaller lexicographical distance from the optimal ordering among other feasible sequences. Since such property still holds true when the procedure is performed on smaller and smaller task subsets, the theorem is proved.

Intuitively, for the whole task set to be schedulable, each task has to be assigned a distinct priority with which it can meet its deadline. Starting from lowest priority, the algorithm tries to find the task with lowest importance that can be assigned the lowest priority. Since there must be one task to be assigned with a given priority for satisfying overall system schedulability, when a task is found schedulable with a priority it is the best task to be at that priority. The procedure can be performed iteratively on the remaining tasks in the same way. The effect is that the final priority ordering, when found, if any, has the minimum lexicographical order among all other schedulable orderings, relative to the desired task ordering.

More formally, consider a priority ordering sequence $S = \langle \tau_1 \tau_2 ... \tau_N \rangle$ with N tasks, which can be written as $S = \langle \psi w \phi \rangle$, where ϕ is a subsequence denoting tasks whose priorities have not been determined and less than K, w the task under schedulability testing for determining priority (priority level K in the UBPO algorithm), and ψ the tasks with their priorities assigned. A desired ordering relation D is assumed to be given over τ so that S^D is the optimal ordering sequence.

For example, let a sequence S be $\langle abcd \rangle$ and $S^D = \langle badc \rangle$. After b is found schedulable with the lowest priority and the schedulability of a in $\langle badc \rangle$ is being checked, the whole sequence can be represented by $\langle \psi w \phi \rangle$, where $\langle \psi \rangle = \langle b \rangle$, $w = a$ and $\langle \phi \rangle = \langle dc \rangle$.

Lemma 2. *The position swapping operation of a task in the subsequence $\langle \phi \rangle$ with w retains the input task ordering relation in $\langle \phi \rangle$.*

Proof. Consider fixing a priority at level K, where size of $\langle\phi\rangle$ is $K-1$, w is the task concerned. There are two cases to consider:

Case 1) w is schedulable at priority level K. If w is schedulable, no swapping has been performed and $\langle\phi\rangle$ has not been changed. Since the original input sequence is ordered by their importance, the importance ordering relation is retained.

Case 2) w is not schedulable with priority level K. If w is not schedulable, the first task to be swapped to the position K is $S[K-1]$ (note the sequence is indexed by K decreasingly from left to right), let's call it τ_s. If τ_s is schedulable at K, $\langle\phi\rangle$ is clearly in accordance with the ordering relation since task $S[K-1]$ certainly has a lower utility than the rest in $\langle\phi\rangle$.

If τ_s is not schedulable at priority level K, observe how it is swapped with $S[K-2]$ as indexed by J now (as in the algorithm). The task $S[K-1]$ and $S[K-2]$ are in the same ordering as in the original sequence where they were at $S[K]$ and $S[K-1]$ - only the task that is of interest is swapped to position K while maintaining the ordering relation in $\langle\phi\rangle$.

When a task's priority is fixed, considering the next task w indexed by K is just another instance of the same decision and swapping procedure mentioned above (Case 1 and 2). Since in each iteration the swapping operation does not concern the tasks in $\langle\psi\rangle$ whose priorities have been fixed, it is easy to see that the lemma holds true as the algorithm carries on fixing priorities from level $K-1$ to 2, after which the algorithm stops when the size of $\langle\phi\rangle$ equals 1. □

Lemma 3. *After a task w is found feasible and its priority fixed, the resulting subsequence $\langle w\phi\rangle$, ignoring $\langle\psi\rangle$, has the smallest lexicographical ordering among all other feasible orderings of $\langle w\phi\rangle$.*

Proof. By Lemma 2 it is known that for any task selected from $\langle\phi\rangle$ which is schedulable at priority level K, $\langle\phi\rangle$ retains the order relation according to the given importance function.

Since the swapping operation only swaps a task with the next lowest utility when one is found not schedulable, task w must be the first feasible task with the lowest utility and $\langle w\phi\rangle$ must have the lexicographically smallest order among all possible ones. □

Assume one final feasible sequence is found by the algorithm, let's call it S^F. If one traverses from the final step of the algorithm to the initial one, the last subsequence of ϕ contains the highest assigned priority task at $S^F[N]$ (with priority 1, the highest in the algorithm), it is guaranteed by the above lemmas that by assigning the task $S^F[N-1]$ (the task w) with its priority the lexicographical distance of $\langle w\phi\rangle$

will be minimized. Similarly, when the sequence ϕ is enlarged to include the next task $S^F[N-1]$, it is also guaranteed by the above lemmas that by assigning the task $S^F[N-2]$ (the new w) its priority the lexicographical distance of the sequence $\langle w\phi\rangle$ will be minimized. The same argument goes on and the lemmas still hold true for the remaining tasks when the lexicographical comparison considers the next task to the left, until all tasks are included. □

As mentioned, similar to the original algorithm, the algorithm takes $O((N^2+N)/2)$ steps where each step only involves performing a schedulability test for one single task, whereas the D&I algorithm requires testing the schedulability of all the lower priority tasks plus the task concerned. The basic (not exact for our task model, but sufficient) response time test [9] for *all* tasks requires pseudo-polynomial time which depends on the number of tasks in the system. The exact test developed for the P-O-E task model has a complexity of $O(2^N)$ time while the tractable sufficient test requires $O(2N)$ for a single task.

7.4 Performance Evaluation

In this section the performance of the D&I and UBPO algorithms is first examined before examining the utility gain, via simulations, from adopting a utility-based approach for assigning priorities to imprecise tasks with higher values.

The primary metric for measuring performance is the total number of exact response time tests, defined as Equations (4.5) and (4.6) for a particular task the algorithms perform in finding the optimal priority ordering. Note that a test is also performed for the task with the highest priority in the system (assumed to be automatically schedulable in the previous section). Further, T_{UBPO} is denoted as the number of response time tests (a busy period calculation) performed by the UBPO algorithm, and, $T_{D\&I}$ as the number of tests used in the D&I algorithm.

As mentioned, the D&I algorithm is not optimal for tasks with offsets because when a priority is tested for a task, the lower priority tasks of which can not be ordered in DMPO with absolute certainty that it is optimal. In other words, if any of the lower priority tasks is not schedulable, there may be other priority orderings for them that are schedulable. Without this optimality, the algorithm will interpret that the priority being tested for the current task is not feasible, and switch to the next task with highest utility among the lower priority tasks.

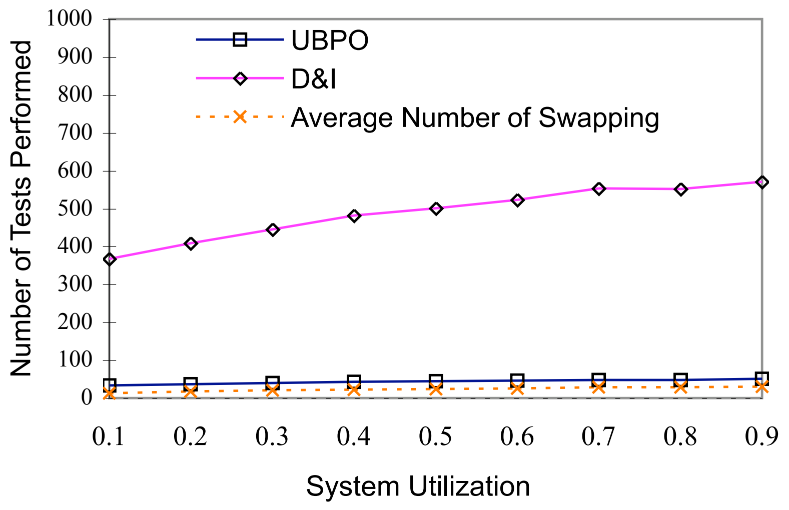

Figure 7.2: Tractable test performance against varying system utilization.

However, since DMPO is generally a good scheduling heuristic and that the focus is in the performance of the two algorithms, it is only when the final priority orderings found by the two are equal are their performance being compared. This evaluation will be valuable because if task offsets are not allowed then the D&I algorithm is still optimal.

Given the utilization of a task system u, a task τ_i (including all prologue, epilogue and normal tasks) is randomly generated with period values spanning across a few orders of magnitude (4 - 7 digit figures) in a roughly uniform way where the value of each digit is chosen randomly (1 - 9 for the most significant digit and 0 - 9 for the rest). Deadlines are chosen to be at about 90% of the period value with small variations such that $C_i \leq D_i \leq T_i$.

Note that the resolution of a simulation tick is one micro-second (μs), thus the tasks generated represent a reasonable period distribution of a real system. All mandatory tasks are executed up to their WCET and they do not hold any resources (no blocking). Once a period is generated, the WCET C of the task is then calculated to provide the desired system utilization. Further, one of the normal tasks will be generated with the skew parameter s as:

$$C = s * u, \tag{7.1}$$

where the WCET of remaining tasks are defined as:

$$C = (1 - s) * u. \tag{7.2}$$

Except the varying parameter in each experiment, all the other parameters are held at default values

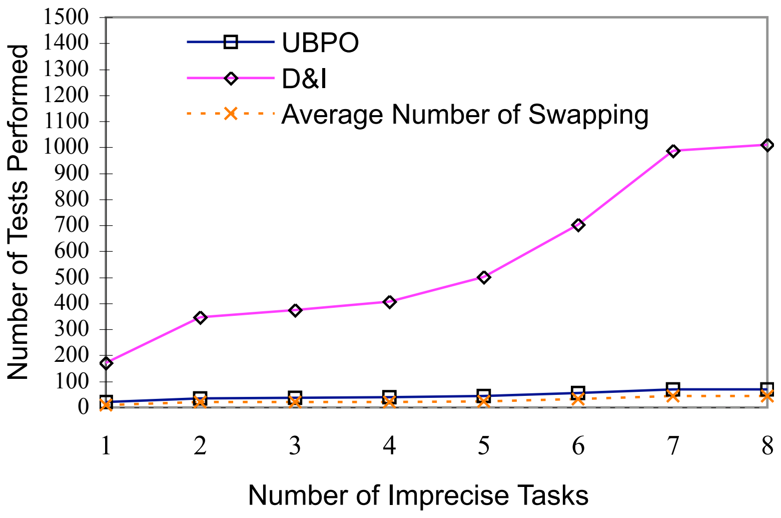

Figure 7.3: Tractable test performance against varying system utilization.

as follows:

- system utilization $u = 0.5$;
- number of imprecise tasks $n = 5$;
- number of normal tasks $m = 10$;
- utilization skew $s = 1/(2n + m)$ such that all tasks equally share u;

Figure 7.2 shows the values of T_{UBPO} and $T_{D\&I}$ against varying number of system utilization. For statistical significance, each data point in the graph represents the average T_{UBPO} and $T_{D\&I}$ over 100 randomly generated task sets with the same set of parameters. A reference line is provided indicating the average number of swapping operations actually performed per task set.

It can be seen that the the number of response time test performed by UBPO, T_{UBPO}, is relatively small and hardly increases as the system becomes more utilized. The value is larger than the number of swapping because at least one testing needs to be done to confirm a valid priority for a task. In fact a few of the task sets generated do not require any priority swapping at all; they are all schedulable with the desired priority ordering.

The same task sets used for UBPO were also used by D&I for producing its performance metric. The number of tests performed increases very steadily upward with higher system utilization, due to the increased likelihood a task may become unschedulable (more interference from other tasks). It can also be seen that there is a big difference in performance between the two algorithms.

Figure 7.3 shows the values of T_{UBPO} and $T_{D\&I}$ against varying number of imprecise tasks. The line of $T_{D\&I}$ shows a more rapid decline in performance (note the scale of y-axis) than in the previous experiment as more imprecise tasks are added into the system. This can be understood by realizing that even in the best case scenario for a system of 20 tasks, UBPO only requires $N = 20$ tests to be performed while D&I will require $(N^2 + N)/2 = 210$ tests - a decisive difference. As more tasks are present in the system, it becomes harder to schedule tasks in their desirable positions. This results in more swapping operations required to be performed (note the slight increase in the reference line) and correspondingly the performance deteriorates very quickly with the D&I approach. This is further compounded by the fact that since the exact test runs in exponential time with regard to the number of imprecise tasks, only a handful number of imprecise tasks can be tested (also see next section).

Note that although the two algorithms minimize lexicographical distance in different directions, it is hard to justify which is better. This is because even being lexicographically smaller, the average difference of task position to their relative optimal position may actually be higher. There is no simple means in differentiating which ordering direction is better. That is, deciding if trying to assign higher priorities to more important tasks is better than trying to assign lower priorities to less important tasks. Note again that only UBPO can work with tasks with offsets, which are required in the task model of iDPS.

In addition, only in 3 out of the 903 generated task sets (thus these 3 task sets are not used) it was observed that the orderings obtained from the two algorithms being different, which is a very small ratio (0.003%). It is also interesting to note that, if the two orderings obtained by the two algorithms are equal, the total number of task priority swappings that are required in finding the optimal ordering by the two algorithms are exactly the same - even though the tasks and the order in which that they are swapped are totally different.

7.5 Employment of Tractable Schedulability Test

The exact test for imprecise computation is computationally prohibitive when there are large number of imprecise tasks present in the system. This motivates a more efficient approach for verifying system schedulability and a tractable test for the task model concerned has been derived in the previous chapter.

Although the test performs extremely well in the response time difference produced compared to

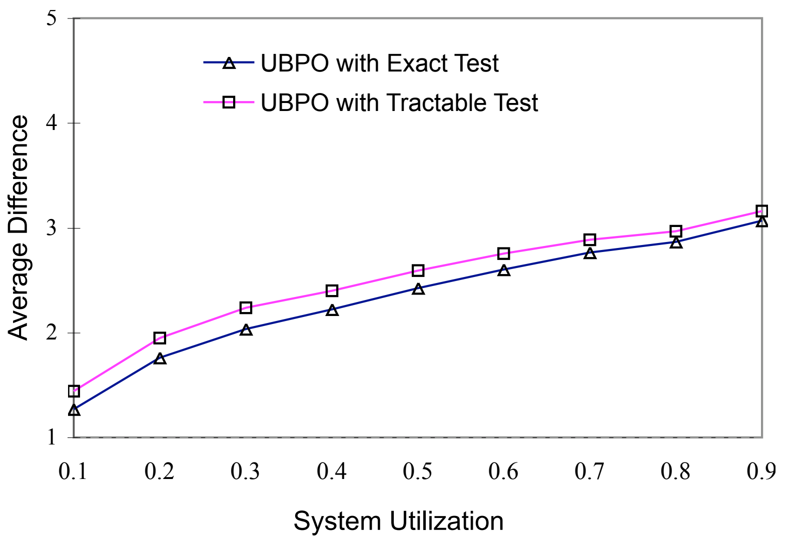

Figure 7.4: Exact and tractable UBPO performance against varying system utilization.

that of the exact test, nevertheless, the UBPO algorithm tries to assign task priorities based on task importance. Since DMPO is no longer employed, tasks are likely to suffer longer interference from other tasks due to the fact that a more urgent task (with shorter deadline) may have to wait for other tasks which have higher utility.

If this happens when the UBPO algorithm uses the tractable test for checking schedulability, the resulting priority ordering may be different from that produced using the exact test. This is because a task may be deemed unschedulable with an assigned priority while it actually is schedulable.

It will be interesting to see how the tractable test affects the results when different parameters are used. The Manhattan distance as a measure is defined as the sum of the absolute distance of each task to its relative position in the optimal priority ordering sequence. More formally, the Manhattan distance function is a function $M : \hat{S} \rightarrow \{0, 1, 2, ...\}$ such that:

$$M(S) = \sum_{\forall i \in \tau} |Pri(S, i) - Pri(S^D, i)|, \tag{7.3}$$

where $Pri(S, i)$ returns the priority position of a particular task τ_i in the priority sequence S so that $M(S^D) = 0$.

For example, with $\tau = \{a, b, c\}$ and a desired ordering $S^D = \langle bac \rangle$, the Manhattan distance of all the possible sequences are listed below:

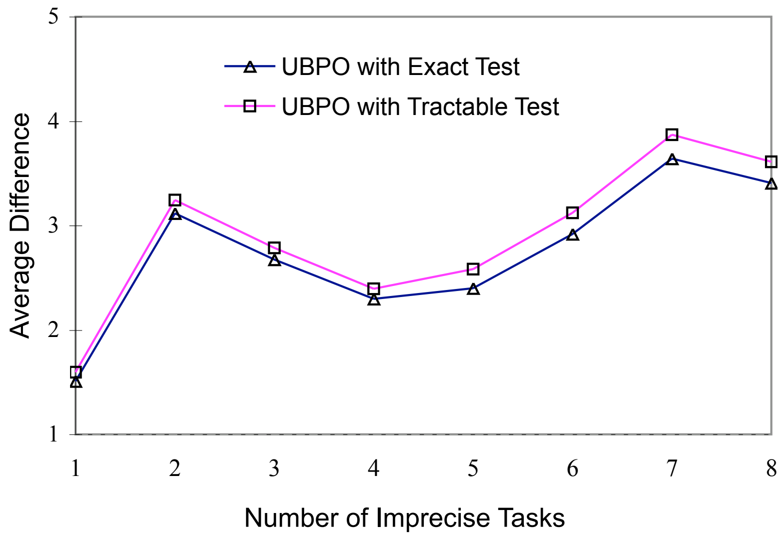

Figure 7.5: Exact and tractable UBPO performance against varying number of imprecise tasks.

$$
\begin{aligned}
M(\langle bac \rangle) &= 0 \\
M(\langle bca \rangle) &= 2 \\
M(\langle abc \rangle) &= 2 \\
M(\langle acb \rangle) &= 3 \\
M(\langle cba \rangle) &= 4 \\
M(\langle cab \rangle) &= 4
\end{aligned}
$$

The metric for measuring the performance of the algorithms is the average Manhattan distance per task in a task set defined as:

$$AM = M/N, \tag{7.4}$$

where N is the number of tasks in the system. Further AM_e is denoted to be the average Manhattan distance per task of the UBPO algorithm performed with the exact test, while AM_t one produced with the tractable test. The parameter settings are similar to the first set of experiments documented in the last section.

Figure 7.4 shows the values of AM_e and AM_t against varying number of system utilization. For statistical significance, each data point in the graph is averaged over 100 randomly generated task sets with the same set of parameters.

It can be observed that as the utilization increases the average Manhattan distance per task also increases, which implies it is harder to obtain an ordering which is as close to the optimal ordering as possible as the system load increases. It can also be seen that the tractable test is tracking the performance of the exact test rather closely.

Figure 7.5 shows the values of AM_e and AM_t against varying number of imprecise tasks. The figure shows that there seems to be no direct correspondence between the performance and the control parameter, since the performance gets worse and then better twice alternatively as more imprecise tasks are present in the system. However, the performance of the tractable test shows great promise (note the scale of the y-axis) in both sets of simulations, with only 0.15 difference in the average Manhattan distance per task set on average. It can be therefore concluded that the tractable test can be safely employed with only little penalty in performance in most cases.

7.6 Summary

The deadline monotonic priority ordering is not optimal for schedulability when offsets are allowed in the task model. Further, when schedulability is satisfied the maximization of system utility by scheduling more important optional components becomes important. This motivates a priority ordering algorithm based on utility, called UBPO. The algorithm is optimal in the sense that it can find any feasible ordering whenever one exists, plus that the lexicographical distance to the optimal ordering (lowest to highest, left to right) is minimized.

Simulations show that the UBPO algorithm is in general much more efficient than the algorithms previously available, where most of the time the final priority orderings found are the same as found by the other optimal algorithm. Where there is a large number of imprecise tasks the tractable schedulability test may have to be introduced in determining task priorities. Experimental data shows that the average Manhattan distance penalty for a large number of randomly generated task sets is only 0.15 on average, proving the usability of the tractable test.

Chapter 8

Conclusion

Supporting real-time AI applications is a challenging problem. By using a simplified case study based on a genuine real-time AI application in the RoboCup competition, the difficulties in developing such programs which can make the best use of system resources are demonstrated. Current approaches, with the restrictions that arise from their task model, scheduling and system support, are insufficient.

In this thesis new results in scheduling imprecise computation based on fixed priority preemptive scheduling (FPPS) were proposed. These include a more general task model for leveraging the problem of algorithm design in addition to the exact schedulability test required to guarantee the mandatory computation, as presented in Chapters 4.

In improving the scheduling of optional components an extended version of dual priority scheduling, called iDPS, for scheduling imprecise computation of the P-O-E structure was presented in Chapter 5. The improvement can be substantial as illustrated by the simulations conducted. In addition, topics with aperiodic tasks scheduling were also discussed.

Since the exact schedulability test can be computationally expensive when the number of imprecise tasks in the system increases, a more tractable, sufficient but not exact, schedulability test was proposed in Chapter 6.

In Chapter 7, the problem of priority ordering based on task utility in the context of iDPS have also been addressed. An optimal algorithm based on one by Audsley was shown to have an extra desirable property in finding the task priorities that are optimal in lexicographical order.

The works presented in this thesis add to the already comprehensive set of theories, scheduling algorithms, and tools based on FPPS, providing a mature technology where real-time AI applications can be suitably supported. It is hoped that, with the basis in FPPS, it can be incorporated into existing real-time operating systems in the near future so that a more general use of imprecise computing can be promoted.

8.1 Future Works

There are a few directions worth investigating. One is the calculation and application of task promotion time; in weakly hard real-time systems [21], two different promotion times may be set for weakly-hard tasks where more time for scheduling optional components can be obtained if a later promotion time is used. Yet when the task is required to meet its timing requirements, a particular promotion selection strategy based on the work by Bernat and Burns [18] may be employed.

Feng and Liu [49] worked on scheduling imprecise tasks with end-to-end timing constraints which allows tasks to have precedence constraints and input errors to tasks. A natural extension to our work will be to allow composite or even hierarchical imprecise tasks [124, 126] where the input quality of a task depends on the output quality of others. Where utilities of optional components of imprecise tasks are characterized by more sophisticated functions, another extension is to allow applications to achieve decision-theoretical control by manipulating their utilities. In this case more advanced scheduling strategies may need to be employed in the scheduling framework. Experimental evaluations using real anytime algorithms with realistic performance profiles will also be valuable.

Recently Davis and Burns [38] formulated the response time analysis for hierarchical scheduling which is a more realistic model of actual systems. A useful future work will be to modify the response time equations in a similar fashion to that by the author for hierarchical imprecise computation. Finally, as with all real-time scheduling problems, the use of multi-core platforms introduces a number of interesting challenges [41]. For example, with the P-O-E model, the prologue component could be executed on the processor attached to the input device, the epilogue component could be executed on the processor attached to the output device, and the optional component executed anywhere on any processor of the platform. Scheduling imprecise tasks on such systems will be a worthy direction to investigate.

List of References

[1] IEEE Std 1003.1d-1999. IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API) Amendment 4: Additional Real Time Extensions [C Language].

[2] A. Aguilar-Soto and G. Bernat. Bi-criteria fixed-priority scheduling in hard real-time systems: Deadline and importance. In the 14th International Conference on Real-Time and Network Systems, Paris, 2006.

[3] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, and J. J. Gutirrez. Fsf: A real-time scheduling architecture framework. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 113–124, 2006.

[4] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS-164, Department of Computer Science, University of York, UK, 1991.

[5] N. C. Audsley. *Flexible Scheduling in Hard Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, UK, 1993.

[6] N. C. Audsley, A. Burns, R. I. Davis, K. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[7] N. C. Audsley, A. Burns, R. I. Davis, and A. J. Wellings. Integrating unbounded software components into hard real-time systems. In S. Natarajan, editor, *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[8] N. C. Audsley, A. Burns, M. F. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[9] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Deadline monotonic scheduling theory. In *Proceedings IFAC/IFIP International Workshop on Real-Time Programming*, pages 55–60, Bruges, Belgium, 1991.

[10] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[11] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. *Computer Systems Science and Engineering*, 8(2):80–89, 1993.

[12] N. C. Audsley, K. Tindell, and A. Burns. The end of the road for static cyclic scheduling. In *Proceedings of the 5th Euromicro Conference on Real-Time Systems (ECRTS 1993)*, pages 36–41, Oulu, Finland, 1993. IEEE Computer Society.

[13] H. Aydin, R. Melhem, and D. Mosse. Incorporating error recovery into the imprecise computation model. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 348, Washington, DC, USA, 1999. IEEE Computer Society.

[14] H. Aydin, R. G. Melhem, D. Mossé, and P. Mejía-Alvarez. Optimal reward-based scheduling of periodic real-time tasks. In *IEEE Real-Time Systems Symposium*, pages 79–89, 1999.

[15] J. M. Banu's, A. Arenas, and J. Labarta. Dual priority algorithm to schedule real-time tasks in a shared memory multiprocessor. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 112.2, Washington, DC, USA, 2003. IEEE Computer Society.

[16] S. K. Baruah and M. E. Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Trans. Comput.*, 47(9):1027–1032, 1998.

[17] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings Real-time Systems Symposium*, pages 328–335, Lisbon, Portugal, 2004. Computer Society, IEEE.

[18] G. Bernat and A. Burns. Combining (n/m)-hard deadlines and dual priority scheduling. In *IEEE Real-Time Systems Symposium*, pages 46–57, 1997.

[19] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *20th IEEE Real-Time Systems Symposium*, Phoenix. USA, Dec 1999.

[20] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems*, 22(1-2):49–75, 2002.

[21] G. Bernat, A. Burns, and A. Llamosí. Weakly-hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, Apr 2001.

[22] M. Boddy and T. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285, 1994.

[23] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Advances in Real-Time Systems*, pages 225–248. Prentice-Hall, 1994.

[24] A. Burns and G. Bernat. Jorvik: A framework for effective scheduling. Technical Report YCS 334, Department of Computer Science, University of York, UK, 2001.

[25] A. Burns, D. Prasad, A. Bondavalli, F. D. 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.

[26] A. Burns and A. J. Wellings. Criticality and utility in the next generation. *Real-Time Systems*, 3(4):351–354, 1991.

[27] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison Wesley, 3rd edition, 2001.

[28] G. C. Buttazzo. *Hard real-time computing systems : predictable scheduling algorithms and applications*. Price/Stern/Sloan Publishers, Los Angeles, CA, USA, 1997.

[29] S. Chen, L. T. X. Phan, J. Lee, I. Lee, and O. Sokolsky. Removing abstraction overhead in the composition of hierarchical real-time systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 81–90, Washington, DC, USA, 2011. IEEE Computer Society.

[30] Cheny, E. Foroughi, F. Heintz, Z. Huangy, S. Kapetanakis, K. Kostiadis, J. Kummeneje, I. Noda, O. Obst, P. Riley, T. Steffens, Y. Wangy, and X. Yiny. Robocup soccer simulator manual 2002. http://sserver.sf.net/docs/manual.pdf.

[31] Y. Chu. On fixed priority preemptive scheduling for imprecise computation. Technical Report YCS-2006-402, Department of Computer Science, University of York, UK, 2006.

[32] Y. Chu. A tractable schedulability test for imprecise computation. Technical Report YCS-2007-414, Department of Computer Science, University of York, UK, 2007.

[33] Y. Chu and A. Burns. On fixed priority scheduling for imprecise computation and the application on deliberative real-time AI systems. In the Workshop on New Trends in Real-Time Artificial Intelligence. The 17th European Conference on Artificial Intelligence, Riva del Garda, Italy, 2006.

[34] Y. Chu and A. Burns. Optimal priority ordering for imprecise computation - a utility-based approach. Technical Report YCS-2007-424, Department of Computer Science, University of York, UK, 2007.

[35] Y. Chu and A. Burns. Supporting deliberative real-time AI systems - a fixed priority scheduling approach. In *Proceedings of the 19th EUROMICRO Conference on Real-Time Systems (ECRTS'07)*, pages 259–268, Pisa, Italy, 2007. IEEE Computer Society.

[36] Y. Chu and A. Burns. Flexible hard real-time scheduling for deliberative AI systems. *Journal of Real-Time Systems, Springer Netherlands*, 40(3):241–263, 2008.

[37] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Trans. Comput.*, 39(9):1156–1174, 1990.

[38] R. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 389–398, 2005.

[39] R. I. Davis. Approximate slack stealing algorithms for fixed priority pre-emptive systems. Technical Report YCS 217, Department of Computer Science, University of York, UK, 1993.

[40] R. I. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. Technical Report YCS 230, Department of Computer Science, University of York, UK, 1994.

[41] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, Oct. 2011.

[42] R. I. Davis, K. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings Real-time Systems Symposium*, pages 222–231. Computer Society, IEEE, 1993.

[43] R. I. Davis and A. J. Wellings. Dual priority scheduling. In *IEEE Proceedings Real-time Systems Symposium*, pages 100–109, Pisa, Italy, 1995.

[44] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, USA, 1988.

[45] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium*, pages 308–319, 1997.

[46] Z. Deng, J. W.-S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *9th Euromicro Conference on Real-Time Systems (ECRTS 1997)*, pages 191–199, 1997.

[47] Z. Deng, J. W.-S. Liu, L. Zhang, S. Mouna, and A. Frei. An open environment for real-time applications. *Real-Time Systems*, 16(2-3):155–185, 1999.

[48] A. J. Dix, R. F. Stone, and H. S. M. Zedan. Design issues for reliable time-critical systems. Technical Report YCS 133, Department of Computer Science, University of York, UK, 1990.

[49] W.-C. Feng and J. W. S. Liu. Algorithms for scheduling real-time tasks with input error and end-to-end deadlines. *IEEE Trans. Software Eng.*, 23(2):93–106, 1997.

[50] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *IEEE Real-Time Systems Symposium*, pages 26–35, 2002.

[51] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, page 199, June 2001.

[52] A. Garvey and V. Lesser. Design-to-time real-time scheduling, special issue on planning, scheduling and control. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1491–1502, 1993.

[53] A. Garvey and V. R. Lesser. A survey of research in deliberative real-time artificial intelligence. *Real-Time Systems*, 6(3):317–347, 1994.

[54] J. Grass and S. Zilberstein. Anytime algorithm development tools. *SIGART Bull.*, 7(2):20–27, 1996.

[55] G. J. Gregory and K.-J. Lin. Building real-time imprecise computations in ada. In *TRI-Ada '90: Proceedings of the conference on TRI-ADA '90*, pages 409–421, New York, NY, USA, 1990. ACM Press.

[56] E. A. Hansen and S. Zilberstein. Monitoring and control of anytime algorithms: a dynamic programming approach. *Artif. Intell.*, 126(1-2):139–157, 2001.

[57] B. Hayes-Roth. Architectural foundations for real-time performance in intelligent agents. *Real-Time Systems*, 2(1-2):99–125, 1990.

[58] K. I.-J. Ho, J. Y.-T. Leung, and W.-D. Wei. Minimizing maximum weighted error for imprecise computation tasks. *J. Algorithms*, 16(3):431–452, 1994.

[59] B. Horling, V. Lesser, R. Vincent, and T. Wagner. The soft real-time agent control architecture. Technical Report TR02-14, Computer Science Dept., University of Massachusetts at Amherst, 2002.

[60] B. Horling, V. Lesser, R. Vincent, and T. Wagner. The Soft Real-Time Agent Control Architecture. *Proceedings of the AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems*, July 2002.

[61] E. J. Horvitz and J. S. Breese. Ideal partition of resources for metareasoning. Technical Report KSL-90-26, Knowledge Systems Laboratory, Stanford University, 1990.

[62] A. E. Howe, D. M. Hart, and P. R. Cohen. Addressing real-time constraints in the design of autonomous agents. *Real-Time Systems*, 2(1-2):81–96, 1990.

[63] X. Huang and A. M. K. Cheng. Applying imprecise algorithms to real-time image and video transmissio. In *IEEE Real Time Technology and Applications Symposium*, pages 96–103, 1995.

[64] D. Hull, W.-C. Feng, and J. W. S. Liu. Operating system support for imprecise computation. In *Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, Cambridge, Massachusetts, Nov 1996.

[65] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, 1992.

[66] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[67] K. B. Kenny and K.-J. Lin. Building flexible real-time systems using the flex language. *Computer*, 24(5):70–78, 1991.

[68] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[69] R. E. Korf. Real-time heuristic search. *Artif. Intell.*, 42(2-3):189–211, 1990.

[70] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 256–267, 1999.

[71] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, 1980.

[72] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *IEEE Proceedings Real-time Systems Symposium*, pages 110–123, 1992.

[73] J. Lehoczky, L. Sha, and J. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings Real-time Systems Symposium*, pages 261–270. Computer Society, IEEE, 1987.

[74] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society, 1989.

[75] V. R. Lesser, J. Pavlin, and E. H. Durfee. Approximate processing in real-time problem solving. *AI Magazine*, 9(1):49–61, 1988.

[76] J. Y. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[77] J. Y.-T. Leung. A survey of scheduling results for imprecise computation tasks. In S. Natarajan, editor, *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[78] J. Y.-T. Leung, V. K. M. Yu, and W.-D. Wei. Minimizing the weighted number of tardy task units. *Discrete Appl. Math.*, 51(3):307–316, 1994.

[79] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *ECRTS*, pages 151–160, 2003.

[80] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[81] J. W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[82] J. W. S. Liu, K.-J. Lin, and S. Natarajan. Scheduling real-time, periodic jobs using imprecise results. In *IEEE Real-Time Systems Symposium*, pages 252–260, 1987.

[83] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, 1991.

[84] C. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[85] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA., 1983.

[86] M. A. Moncusi, A. Arenas, and J. Labarta. Improving energy saving in hard real time systems via a modified dual priority scheduling. *SIGARCH Comput. Archit. News*, 29(5):19–24, 2001.

[87] D. J. Musliner, E. H. Durfee, and K. G. Shin. CIRCA: A cooperative intelligent real time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1561–1574, 1993.

[88] D. J. Musliner, J. A. Hendler, A. K. Agrawala, E. H. Durfee, J. K. Strosnider, and C. j. Paul. The challenges of real-time AI. *Computer*, 28(1):58–66, 1995.

[89] J. Paul K. Harter. Response times in level-structured systems. *ACM Trans. Comput. Syst.*, 5(3):232–248, 1987.

[90] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[91] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *IEEE Real-Time Systems Symposium*, pages 3–14, 2001.

[92] M. A. Rivas and M. G. Harbour. MaRTE OS: An Ada kernel for real-time embedded applications. In *Ada-Europe*, pages 305–316, 2001.

[93] S. Russell and E. Wefald. *Do the right thing: studies in limited rationality*. Artificial intelligence. MIT Press, Cambridge, Mass, 1991.

[94] S. J. Russell. Principles of metareasoning. *Artif. Intell.*, 49(1-3):361–395, 1991.

[95] S. J. Russell and P. Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, Upper Saddle River, N.J., 2nd international edition, 2002.

[96] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd ed.* Pearson Education, 2002.

[97] S. J. Russell and S. Zilberstein. Composing real-time systems. In *IJCAI Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 212–217, Sydney, Australia, 1991.

[98] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed priority scheduling. In *ECRTS*, pages 173–181, 2002.

[99] L. Sha, T. F. Abdelzaher, K.-E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[100] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *IEEE Real-Time Systems Symposium*, pages 181–191, 1986.

[101] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[102] W. Shih, J. W. S. Liu, J. Chung, and D. W. Gillies. Scheduling tasks with ready times and deadlines to minimize average error. *SIGOPS Oper. Syst. Rev.*, 23(3):14–28, 1989.

[103] W.-K. Shih and J. W. S. Liu. On-line scheduling of imprecise computations to minimize error. *SIAM J. Comput.*, 25(5):1105–1121, 1996.

[104] W.-K. Shih, J. W. S. Liu, and J.-Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM J. Comput.*, 20(3):537–552, 1991.

[105] H. A. Simon. *Models of Bounded Rationality*. MIT Press, 1982.

[106] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[107] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[108] J. A. Stankovic. Real-time computing systems: the next generation. In *Tutorial: hard real-time systems*, pages 14–37. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[109] J. A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253, 2004.

[110] J. A. Stankovic and K. Ramamritham. Editorial: What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990.

[111] J. A. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, 1991.

[112] J. A. Stankovic, K. Ramamritham, D. Niehaus, M. Humphrey, and G. Wallace. The Spring system: Integrated support for complex real-time systems. *Real-Time Systems*, 16(2-3):223–251, 1999.

[113] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, Jan 1995.

[114] J. K. Strosnider and C. J. Paul. A structured view of real-time problem solving. *AI Mag.*, 15(2):45–66, 1994.

[115] A. Terrasa, A. García-Fornes, and V. J. Botti. Flexible real-time linux*: A flexible hard real-time environment. *Real-Time Systems*, 22(1-2):151–173, 2002.

[116] T.-S. Tia, J. W. S. Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Real-Time Systems*, 10(1):23–43, 1996.

[117] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical Report YCS-92-182, Department of Computer Science, University of York, UK, 1992.

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

[119] R. Vincent, B. Horling, V. Lesser, and T. Wagner. Implementing Soft Real-Time Agent Control. *Proceedings of the 5th International Conference on Autonomous Agents*, pages 355–362, June 2001.

[120] S. V. Vrbsky and J. W. S. Liu. An object-oriented query processor that produces monotonically improving approximate answers. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 472–481, Washington, DC, USA, 1991. IEEE Computer Society.

[121] P. H. Winston. *Artificial intelligence (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1984.

[122] J. Yen and S. Natarajan. A decision-theoretic treatment of imprecise computation. In S. Natarajan, editor, *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[123] W. Zhao, C. C. Lim, J. W. S. Liu, and P. D. Alexander. Overload management by imprecise computation. In S. Natarajan, editor, *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[124] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, University of California at Berkeley, 1993.

[125] S. Zilberstein and S. Russell. Approximate reasoning using anytime algorithms. In S. Natarajan, editor, *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[126] S. Zilberstein and S. Russell. Optimal composition of real-time systems. *Artif. Intell.*, 82(1-2):181–213, 1996.