Value-Driven Multi-Class Overload Management in Real-Time Database Systems

Jörgen Hansson

Department of Computer Science
University of Skövde
Box 408
SE-541 28 Skövde, Sweden

jorgen.hansson@ida.his.se

September 1999

ISBN 91-7219-542-8
ISSN 0345-7524

Printed by UniTryck, Linköping 1999

We must not believe the many, who say that only free people ought to be educated, but we should rather believe the philosophers who say that only the educated are free.
- Epictetus

Abstract

In complex real-time applications, real-time systems handle significant amounts of information that must be managed efficiently, motivating the need for incorporating real-time database management into real-time systems. However, resource management in real-time database systems is a complex issue. Since these systems often operate in environments of imminent and transient overloads, efficient overload handling is crucial to the performance of a real-time database system.

In this thesis, we focus on dynamic overload management in real-time database systems. The multi-class workload consists of transaction classes having critical transactions with contingency transactions and non-critical transactions. Non-critical transaction classes may have additional requirements specifying the minimum acceptable completion ratios that should be met in order to maintain system correctness. We propose a framework which has been implemented and evaluated for resolving transient overloads in such workloads.

The contributions of our work are fourfold as the framework consists of (i) a new strategy and (ii) a new scheduling architecture for resolving transient overloads by re-allocating resources, (iii) a value-driven overload management algorithm (OR-ULD) that supports the strategy, running in $O(n \log n)$ time (where n is the number of transactions), and (iv) a bias control mechanism (OR-ULD/BC). The performance of OR-ULD and OR-ULD/BC is evaluated by extensive simulations. Results show that, within a specified operational envelope, OR-ULD enforces critical time constraints for multi-class transaction workloads and OR-ULD/BC further enforces minimum class completion ratio requirements.

Acknowledgements

I wish to express sincere gratitude to all those who have helped me in various ways. Foremost I would like to thank my advisor group for their excellent guidance and their support: my supervisor Prof. Sten Andler for giving invaluable advice and direction on the overall structure and form of the research, Prof. Sang Son for giving many insightful comments and suggestions, and Prof. Tore Risch for reading my manuscripts.

I am indebted to Prof. Jack Stankovic for his valuable advice and the interesting discussions I had with both him and Prof. Sang Son during my time at University of Virginia. Moreover, I am thankful to Ming Xiong who with Prof. Stankovic introduced and guided me through the secrets of the RADEx simulator.

The research forms a part of the DeeDS project and as such has benefited from the experience of other project members; Dr. Bengt Eftring, Joakim Eriksson, and Jonas Mellin. Egir Örn Leifsson and Ragnar Birgisson have helped proofread the manuscript.

Behind every thesis seems to be at least one person who has been particularly influential in directing the student towards research. In my case this was Stig Emanuelsson, head of department, who has continuously supported me and provided great opportunities for my development as a researcher. Special thanks to Dr. Brian Lings, supervisor for my M.Sc. project, who inspired me to pursue doctoral studies. My gratitude extends to vice-chancellor Lars-Erik Johansson, for his encouragement and support. I am also truly grateful to Lillemor Wallgren for all her help with administrative matters and to Ivan Rankin who checked

the manuscript for linguistic errors.

I thank my friends Mikael Berndtsson, Karin Lundberg, and Björn Olsson with whom I have shared many great moments and much laughter during this time.

Finally, my particular thanks to my parents for their constant yet undemanding encouragement throughout my academic career, and to my sister Camilla and her family for being a source of joy and support. This thesis also represents a response to their most frequent question "When will you be finished?" (this question was most often followed by "Again, what is the focus of your research?").

Jörgen Hansson

September 1999

List of Publications

- J. Hansson and S.H. Son, Overload Management in Real-Time Database Systems, To appear in Real-Time Database Systems: Issues and Design, Artech Publishers, 1999.

- J. Hansson and S.F. Andler, System Framework for Active Real-Time Database Systems, To appear in Real-Time Database Systems: Issues and Design, Artech Publishers, 1999.

- J.A. Stankovic, S.H. Son, and J. Hansson, Misconceptions about Real-Time Database Systems, IEEE Computer, 32:6, June 1999.

- J. Hansson, S.H. Son, J.A. Stankovic and S.F. Andler, Dynamic Transaction Scheduling and Reallocation in Overloaded Real-Time Database Systems, Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98), Hiroshima, Japan, IEEE Computer Society Press, 1998.

- S.F. Andler and J. Hansson (eds), Proceedings of the Second International Workshop on Active, Real-Time, and Temporal Database Systems (ARTDB-97), Como/Milan, Italy, Springer-Verlag Lecture Notes, 1998.

- S.F. Andler, J. Hansson, J. Eriksson J. Mellin, and B. Eftring, Overview of the DeeDS Architecture, Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'98), Orlando, Florida, 1998.

- J. Hansson and M. Berndtsson, Active Real-Time Database Systems, In Active Database Systems, Norman Paton (editor), Springer-Verlag, 1998.

- J. Mellin, J. Hansson, and S.F. Andler, Refining Design Constraints of Applications in DeeDS, Chapter 18, In Real-Time Database Systems. A. Bestravos, K-J Lin, and S. H. Son (eds), Kluwer Academic Publishers, 1997.

- M. Berndtsson and J. Hansson, Workshop Report: The First International Workshop on Active and Real-Time Database Systems (ARTDB-95), SIGMOD Record, Special Section on Real-Time Databases, March 1996.

- J. Mellin, J. Hansson, and S.F. Andler, Deriving Design Constraints from a System Services Model for a Real-Time DBMS, RTDB'96 Workshop, Newport, California, USA, 1996.

- S. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson and B. Eftring, DeeDS Towards a Distributed Active and Real-Time Database System, SIGMOD Record, Special Section on Real-Time Databases, March 1996.

- M. Berndtsson and J. Hansson (editors), Active and Real-Time Database Systems (ARTDB-95), Proceedings of the First International Workshop on Active and Real-Time Database Systems, Skövde, Sweden, 9-11 June 1995, 284pp, ISBN 3-540-19983-7, Springer Verlag (London) Ltd.

- M. Berndtsson and J. Hansson, Issues in Active Real-Time Databases Proceedings of International Workshop on Active and Real-Time Database Systems ARTDB-95, 1995.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

''Where shall I begin, please your Majesty?'' he asked.
''Begin at the beginning'', the King said, very gravely,
''and go on till you come to the end: then stop.''
- L. Carroll, Alice's Adventures in Wonderland

This thesis proposes a framework for dynamically resolving transient overloads in real-time database systems, yielding predictable behavior and graceful performance degradation during transient overloads. The framework consists of a strategy and a scheduling architecture. The strategy to resolve transient overloads is to scrutinize resource allocations of admitted transactions and determine the cost of releasing sufficient resources for admitting a new transaction. Further, the decision on whether to admit an original transaction, admit a contingency transaction, or reject the transaction is balanced against the cost of releasing resources among already admitted transactions. A scheduling architecture and a strategy, and an algorithm (OR-ULD) that supports this strategy, have been defined and evaluated. The thesis shows that the strategy of resolving overloads by dynamically reallocating resources among admit-

ted transactions yields better performance than other dynamic strategies employing admission control only. The proposed algorithm, denoted OR-ULD, runs in $O(n \log n)$ time, where n is the number of transactions.

The chapter starts with an overview (section 1.1). The succeeding two sections give a concise introduction to real-time systems (section 1.2) and real-time database systems (section 1.3) and how they are related. The chapter concludes with a high-level description of the research problem high-lighting the research contributions (section 1.4), followed by a description of how the thesis is structured (section 1.5).

1.1 Overview

Studying complex computing systems, e.g., air traffic control and military mission control, shows that many systems must be capable of handling extensive amounts of information while supporting real-time requirements. These systems require efficient techniques for representation and storage of data objects, and time-cognizant methods for accessing and manipulating data objects. These requirements indicate there is a need for *real-time database systems*, i.e., database systems with predictable transaction processing.

There are many misconceptions suggesting that conventional databases are adequate for real-time applications [SSH99]. Unfortunately, conventional database systems do not enforce real-time requirements due to the inherently incompatible, and sometimes contradictory, design criteria. Conventional database systems are optimized for minimizing average response time and maximizing transaction throughput. Moreover, conventional database systems introduce several sources of unpredictability making them inappropriate for real-time applications (see [Ram93] for a discussion).

The development and construction of real-time database systems present challenging research problems that have attracted a lot of attention [rtd88, BH95b, rtd96, BLS97, BW97a, AH98]. One complex issue in real-time database systems is resource management, and particularly

overload handling, since real-time database systems often operate in situations of imminent overloads [Loc96, SSH99]. Hence, it is important that the system has, in addition to an overload detection mechanism, an overload mechanism and a policy defining how transient overloads should be resolved.

A *transient overload* denotes a system state that lasts for a finite duration where the resource demand as defined by the transaction workload exceeds the available resource capacity of the system. As a consequence the system fails to fully enforce its consistency requirements (temporal, logical, external, etc.). In order to minimize the effects of a transient overload, it is desirable that the system manages its resources such that system performance degrades gracefully. While several scheduling policies, e.g., Earliest Deadline First and Least Slack, produce optimal results under non-overload conditions, they generally exhibit poor performance in overload situations.

Resource management in real-time database systems can be divided into several activities: *admission control* — determining which transactions should be granted system resources; *scheduling* — determining the execution order of admitted transactions; and *overload management* — determining how to resolve transient overloads. While scheduling focuses on *when* to execute transactions, admission control and overload management focus on selecting *which* transactions should be allowed to execute. Overload management policies can in general be classified into prevention policies or resolution policies. A typical prevention policy is, for example, to limit the number of transactions that arrive to the system. In contrast, resolution policies allow transient overloads to occur, in which case the overload resolution policy attempts to remove the transient overload before the system malfunctions. Possible policies for overload resolution include: transaction rejection, transaction termination, transaction migration, deferred transaction execution, transaction replacement, and partial transaction execution.

Transaction rejection implies that transactions are rejected upon their arrival. In contrast, *transaction termination* implies that overloads are resolved by aborting transactions eligible for execution, i.e., transactions

that have been admitted to the system. Approaches in this category include [BSS95, DMK+96, BN96] (see sections 6.2 and 6.4).

Transaction migration is applicable in distributed database systems, where a pending overload at a node can be resolved by migrating transactions to a node with enough spare capacity to ensure their timeliness (ability to meet their time constraints). In a database context transactions are normally migrated to database nodes having local copies of the required data, i.e., the database is, partially or fully, replicated. Migrating transactions are in general costly due to the additional communication costs associated with selecting a destination node which normally includes information about the workload on the destination node being exchanged, and migrating the transaction to the destination node. Early work on cost efficient load balancing has been reported in [SRC85].

Deferred transaction execution implies that execution and completion of transactions having a deadline tolerance, e.g., soft deadline transactions, are postponed. Transient overloads are, per definition, finite in length and deferring the execution of non-critical transactions outside the overload interval will decrease the resource demand in the critical overload interval. Hence, in the critical overload interval resources are granted to transactions with no deadline tolerance.

Transaction replacement is applicable when transactions have one or several alternative transactions having lesser resource requirements than the original transaction. Normally the alternative transactions produce satisfactory results but with lower quality than the original transaction, or they compensate original transactions that cannot complete successfully. One of the approaches in this category is presented in [LC86, Che94, Nag97, BN97, BN96] (see sections 6.5 and 6.5.3).

Partial transaction execution, e.g., imprecise computation [LLS+91] where transactions are decomposed into one mandatory and one optional task, suggests that only mandatory parts are executed during transient overloads. Algorithms proposed for imprecise workloads include [SLC89, SLC91, SL92, ZLLA95, LRS+98, KSCJ98]) (see sections 6.5 and 6.5.1).

These prevention policies reduce the resource demand, requested by transactions, to a level that can be handled by the system. Current state-of-the-art scheduling algorithms featuring overload tolerance normally employ only one of the listed policies.

This thesis proposes a new framework for resolving transient overloads, using multiple resolution policies, in real-time databases.

1.2 Real-Time Systems

In *real-time systems*, the correctness of a result depends not only on the logical result of the computation, but also on the time at which the result is produced [BW97b]. Hence, system correctness depends on both the functional and temporal behavior of the system execution. The executable entities in real-time systems are normally referred to as tasks. Real-time database systems represent one class of real-time systems, where the executable entities are referred to as transactions. For reasons of simplicity, time-constrained tasks or transactions are throughout the thesis referred to as transactions.

Real-time systems can be categorized by the penalty imposed on the real-time system in case of a violated time constraint. Depending on the criticality of the transactions involved, three types of real-time systems can be discerned. In a *hard* real-time system, time constraints must always be met. These deadlines are critical since lateness has severe or, in some cases, catastrophic consequences. Two distinct sub-classes can be formed, based on whether continued system execution can be performed in the presence of a failure. A deadline that is imperative to meet is sometimes referred to as *hard critical*, and a transaction imposes an infinite penalty on the system if its deadline is missed. In contrast, a transaction with a *hard essential* deadline imposes a finite penalty on the real-time system in the case when it is missed. In a *soft* real-time system, the deadline should be met, but it is acceptable if transactions occasionally miss their deadlines and complete after their deadline. In a *firm* real-time system, deadlines should also be met, but it is acceptable if

transactions occasionally miss their deadlines. In contrast to soft deadline transactions, however, no lateness is acceptable and, hence, tardy transactions are aborted in order to reduce resource wastage.

It is generally accepted that real-time systems of the next generation are going to operate in non-static environments since the working environment will constantly change [Sta88a, Sta88b]. A real-time system must therefore be able to work in a non-deterministic and dynamic environment with limited a priori knowledge about future events and transactions arriving to the system. The workload imposed on the system contains transactions with different types of deadline criticality. These real-time systems are referred to as *multi-class real-time systems.*

1.3 Real-Time Database Systems

The next generation of real-time systems and applications will handle substantial amounts of information that require sharing, distribution, and replication. Data accesses must not jeopardize any time constraints of transactions [Ram93]. In conventional database systems the description of data is stored in the database and is available as a resource. Maintenance of database consistency is guaranteed by enforcing transaction atomicity, consistency, isolation, and durability (the ACID-properties) using concurrency control techniques. Normally, conventional database systems are designed to maximize transaction throughput. However, real-time computing is not high-performance computing [Sta88a]. Instead, timeliness is the paramount goal in real-time database systems. Real-time database systems differ from conventional database systems in several ways. First, in real-time database systems, not all data may be permanent; some data may be temporary. Second, transactions have time constraints which require that transaction execution enforces the constraints, requiring that scheduling and concurrency control algorithms are time cognizant. Third, meeting the deadline of a transaction might be more important than an exact result. Therefore, temporal inconsistency is acceptable in some situations in order to guarantee that

the deadline is met. That is, the correctness of the result is traded for timeliness by relaxing consistency.

Scheduling in multi-class real-time systems poses new problems. Complex real-time systems usually cannot have a single scheduling policy due to the size of the system, the vastly different requirements of various sets of transactions, and the different metrics used for different functions or subsystems over single-class scheduling systems [Sta95]. In this work, time constraints are represented with value-functions, and a combined deadline- and value-driven approach is taken. The focus is on dynamic real-time scheduling for a single processor where there is an additional processing element dedicated for scheduling and monitoring. The underlying workload assumption is that the set of critical transactions, based on the worst case execution time of the corresponding contingency actions, is schedulable. Hence, there exists an execution order such that the deadlines of the contingency actions of the corresponding critical transactions are met.

1.4 Research Contributions

This thesis focuses on how transient overloads, in real-time database systems, can be resolved dynamically without jeopardizing the predictability (and thus the timeliness) of the system. Our main contributions are fourfold. In this thesis we introduce a framework for resolving transient overloads. The framework consists of (i) a new strategy for resolving transient overloads, and (ii) a novel scheduling architecture that includes dynamic admission control, a transaction scheduler, an overload resolver, and a dispatcher. (iii) A new overload resolution algorithm (that implements the strategy), denoted OR-ULD, capable of removing transient overloads in real-time database systems with complex transaction workloads has been implemented and evaluated. OR-ULD has been equipped with (iv) a bias control mechanism (/BC) that enforces minimum completion ratio requirements for transaction classes. The resulting scheduling model forms the basis for the real-time transaction scheduler in the DeeDS

prototype [AHE+96].

Transactions have critical deadlines or non-critical deadlines. In addition, critical transactions have contingency transactions that can be invoked in case of overloads, replacing the original transaction. Our algorithm attempts to maximize the sum, over all transactions, of utility contributed to the system. More importantly, as our performance study shows, the algorithm enforces the time constraints of critical transactions.

Transient overloads are detected by the admission controller which invokes the overload resolver. The overload resolver develops a plan to de-allocate enough resources among the admitted transactions in order to be able to admit the new transaction, and determines whether it is advantageous to carry out the plan or not.

The OR-ULD algorithm has the following characteristics:

- it operates in transient overload conditions, i.e., it is only invoked in the case where a transient overload is detected;
- it resolves transient overloads by (i) scrutinizing current resource reservations made by admitted transactions and, if necessary, (ii) revoking previous reservations;
- it ensures the timeliness of critical transactions and gracefully degrades system performance during transient overloads;
- it enables the use of multiple overload resolution strategies, e.g., in our case transient overloads are resolved by controllably dropping non-critical transactions and selectively replacing critical transactions with contingency transactions;
- it separates overload management from scheduling and can therefore be combined with various scheduling policies (given that the admission controller is able to detect transient overloads and can indicate in which time interval the overload is occurring); and
- it handles multi-class transaction workloads, where classes are discriminated by their criticality.

A performance study is accomplished via simulation. The OR-ULD algorithm is compared with the traditional EDF with an admission controller, and a baseline algorithm. The simulation results indicate that OR-ULD outperforms these algorithms under a wide range of workload conditions. Moreover, OR-ULD yields controlled overload behavior and enforces timeliness of critical transactions. Further, simulation results show that OR-ULD/BC biases the execution of transaction classes based on their minimum completion ratio requirements and enforces the requirements within a specified operational envelope.

1.5 Thesis Outline

The thesis follows the following outline. Chapter 2 provides a background description and defines the basic terminology used throughout this thesis. In chapter 3 a detailed description of the real-time scheduling problem is given, and assumptions and simplifications are introduced. In chapter 4 we introduce the overload resolution algorithm OR-ULDand the bias control mechanism (/BC). Chapter 5 reports on the performance of OR-ULD and OR-ULD/BC, comparing the results to a set of reference algorithms. The chapter also includes a description of the simulation environment. Chapter 6 contrasts the proposed framework and overload resolution strategy with current state of the art research in overload management in real-time database systems. Finally, contributions and conclusions, together with some directions for future research, are presented in chapter 7.

Chapter 2

Background to Real-Time Database Systems

This chapter introduces the terminology used throughout the thesis. First, a background to real-time systems (section 2.1) and real-time database systems (section 2.2) is presented. A discussion on real-time scheduling follows (section 2.3). This work has been carried out as a part of the DeeDS project in which a distributed active real-time database system is being built. Therefore, this chapter concludes with a description of the role of active real-time database systems in real-time systems (section 2.4), a background to active real-time database systems (section 2.5), and an overview of the DeeDS architecture and its components (section 2.6).

2.1 Real-Time System Preliminaries

When comparing real-time systems with other computer systems, they differ significantly in design goals, requirements, and implementation. The fundamental difference is the importance of time in different respects. With respect to time, computer systems have traditionally been designed and optimized to guarantee high throughput and minimize the response time. New applications requiring prompt response at certain

time-points imply that the strict notion of time and timeliness are addressed and handled in the underlying computer system. This system we call a real-time system. Several good definitions of real-time systems exist. Lawson [Law92] defines a real-time system to be:

> "... a system that assures that controlled activities "progress" and that stability is maintained and further, that the values of outputs **and** the time at which the outputs are produced are important to the proper functioning of a system."

A more general definition is presented by Young, who defines a real-time system to be [You82]:

> "... any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period."

Hence, a real-time system is a system that, in order to maintain external/internal correctness, has to respond to input stimuli and produce logical results of computation within a finite and sufficiently small time bound. It is often stated in the literature that real-time systems must feature *predictability*. Stankovic and Ramamritham [SR90] describe predictability as the ability to, in some way, show that the system meets the specified requirements under various conditions, e.g., failures and workloads, the system is expected to work under. A similar view is given by Le Lann [LL91], where predictability means that the system behaves according to the specifications. Le Lann argues that predictability is the likelihood that assumptions made at specification and design time are not violated at run-time, and that the system behaves as anticipated whenever run-time conditions match specification and design assumptions. If the system behavior deviates from the specification, the system acts under conditions that have not been foreseen, jeopardizing the predictability, which may have severe consequences.

Timeliness is in some safety-critical applications an imperative requirement, leaving no room for lateness. However, the applications may have

weaker requirements of the quality of the result. Hence, the logical correctness can be traded for timeliness, implying that the correctness of a real-time system is a function of (i) the logical correctness of the computation, and (ii) the timeliness of result delivery. The most important property of a real-time system is predictability, i.e., the functional and temporal behavior is deterministic to such an extent that the system specification is satisfied [Sta88a]. Given this, it is not hard to see that real-time computing is not an issue of high-performance computing.

Real-time systems are typically embedded systems, where the system can, on a high-level, be decomposed into one *controlled* system, representing the environment in which the real-time system acts, and one *controlling* system, interfaced with (intelligent) sensors and actuators. Moreover, a human operator is connected to the controlling system for monitoring purposes. The main characteristics of a real-time system are the following: (i) they interact with their environment; (ii) they react to stimuli in the environment; and (iii) the correctness of the result not only depends on the functional result, but also *when* the result is delivered.

2.2 The Role of RTDBs in Real-Time Systems

When studying different and especially complex applications, several common characteristics and requirements appear. First, massive amounts of data are often retrieved by the controller system from the environment and they must be computed and stored. Second, in some applications the amounts of data needed to produce an output, given a specific input, are significant, i.e., the size of database used for making a decision is significant. A typical application is, for example, geographic information systems in airplanes. Third, systems are responding to physical events, i.e., the systems incorporate reactive behavior. Fourth, the systems act in environments with dynamic elements, implying that dynamic and adaptable behavior is desirable and sometimes a necessary property of real-time systems. Fifth, the real-time system must be able to handle complex workloads since tasks differ in strictness, tightness,

arrival patterns, etc. A complex real-time system needs database functionality (efficient storage and manipulation of data), and support for specifying reactive behavior. Moreover, it must be able to adapt to its environment in order to cope with dynamic situations.

Conventional database systems are not suitable as repositories for real-time systems due to the inherently different design goals and criteria. Conventional databases are designed to maximize throughput and minimize the average response time while maintaining database consistency. Hence, real-time guarantees cannot be given, and therefore time-cognizant techniques for transaction processing must be developed, where transactions have time constraints. A system that supports this is referred to as a *real-time database system (RTDB)*. Conventional and real-time database systems differ in several respects. As mentioned, the performance metrics are different, and in a real-time database system the metric is usually to maximize the *completion ratio* or to minimize the number of missed deadlines. Hence, the scheduling algorithm must be time-cognizant. Moreover, concurrency control protocols are used to increase the concurrency of transaction execution and still maintain database consistency. Where the concurrency control algorithm and the scheduler are seen as autonomous entities, decisions made of the former algorithm can override scheduling decisions made by the scheduler [YWLS94]. Therefore both the concurrency control algorithm and the scheduler in a real-time database system must be time-cognizant, ensuring timeliness. Although database consistency is desirable, in hard real-time database systems it can be that timeliness must be traded for consistency. Hence, the timely delivery of a result of lower quality or accuracy is preferred to having an exact result delivered too late. Data-access conflicts are also handled differently. In conventional databases data access conflicts are resolved based on fairness or resource consumption, while in real-time database systems the preference tends to be criticality of the transaction [YWLS94].

For more material regarding real-time database management, see the work of Graham [Gra92], Kao and Garcia-Molina [KGM92], Ramamritham [Ram93] and Yu et al. [YWLS94], Eriksson [Eri97], Özsoyoğlu

and Snodgrass [OgS95], and others [rtd88, rtd96, BLS97, BW97a, LK99].

2.2.1 Categorization of Real-Time Systems

Real-time systems, in their various subtleties, can be categorized according to different aspects: the granularity of a deadline; the laxity of tasks; the strictness of a deadline; reliability; the size of a system and degree of coordination; and the environment in which the system operates [SR90].

By studying the severity of the consequences when the real-time system's behavior is temporally incorrect, i.e., time constraints are not satisfied, the importance of timeliness can be quantified and, hence, used for categorizing real-time systems. By analyzing the strictness of deadlines, two fundamentally different types of systems exist. In *hard real-time systems* it is an imperative requirement that deadlines are met, due to the catastrophic and costly consequences caused by delays. In *soft real-time systems* tasks should execute to completion and their deadlines should be met, but can be missed occasionally without violating system correctness. A special case of soft real-time systems is a *firm real-time system*, in which no tasks are executed after the deadline. Although soft transactions can occasionally deliver results late and firm deadlines can occasionally be missed, there is normally an upper limit on tardiness and the number of misses within a defined interval [BW97b].

Hard real-time systems can be classified further. In *hard critical real-time systems* it is absolutely imperative that deadlines are met and failing to meet these causes system failure. In contrast, in *hard essential real-time systems* a missed deadline has severe consequences but system services, although at a degraded level, can still be provided satisfactorily.

2.3 Real-Time Scheduling Preliminaries

The task of a real-time system is to provide a response, within a predictable and finite bounded time, to an input stimulus where the response consists of providing a computational result. The code for computing the

result is encapsulated in a transaction with an associated time constraint. Given a transaction workload, the execution order of transactions must be determined in order to ensure time constraints. This task is carried out by a scheduler. The transaction execution order generated by the scheduler is referred to as a *schedule*.

In a *feasible schedule* transactions are executed in an order such that the temporal behavior of the real-time system is correct, i.e., the deadlines of all transactions are met. A scheduler is said to be *optimal* given that it always finds a feasible schedule when there is one.

Since no lateness is allowed in hard real-time systems, there is an implicit requirement that scheduling decisions should be based on the worst-case execution time of transactions in order to guarantee time constraints and enhance the predictability of the real-time system [BW97b, BW90, XP90].

The violation of a soft deadline does not cause severe damage unless violations occur frequently. However, as Burns [BW97b] points out, typically soft deadlines have an upper limit of misses within a defined interval. Soft deadlines allow for some tardiness, i.e., late delivery of the result is acceptable and is better than no result. However, there may be an upper limit on tardiness. A firm deadline is a special case of a soft deadline. However, in contrast to soft deadlines, no lateness of result delivery is allowed, implying that no further execution of a firm deadline transaction should be carried out after the deadline.

A *precedence constraint* exists if there is a constraint on the execution order between two transactions. Formally, a precedence constraint $\tau_1 << \tau_2$ expresses that τ_1 must complete before τ_2 starts to execute. If transaction τ_1 fails to complete, e.g., due to transient overload, transaction τ_2 will not start.

Transaction scheduling is either *preemptive* or *non-preemptive*. Executing a non-preemptable transaction implies that the real-time database system must execute the transaction to completion. In contrast, in preemptive scheduling the current executing transaction may be preempted in order to favor a newly arrived and more important transaction. De-

pending on the characteristics of the transactions, different forms of interleaving are adopted, e.g., preemption-abort, preemption-restart, and preemption-resume. The methods differ in the type of operation that should be performed on the preempted transaction. The *preemption abort* policy implies that currently running transactions are aborted upon preemption. The *preemption restart* method aborts transactions upon preemption but transactions are then restarted. *Preemption resume* implies that a preempted transaction is suspended and its execution is continued upon the completion of the preempting transaction.

There are two distinct approaches for solving real-time scheduling decisions: static scheduling and dynamic scheduling. The main difference between the two approaches is at what time scheduling decisions are made in relation to the start time of the real-time system. In the following we elaborate on the pros and cons of each one of these approaches.

A scheduling algorithm is considered to be *static* if the search for a feasible schedule is carried out before system run-time. Hence, a complete time-map is developed, and the purpose of the scheduler at run-time is to dispatch transactions according to the schedule. Static scheduling, also known as pre-runtime scheduling, requires that most information about the transaction characteristics, such as release times, deadlines, worst-case-execution times, precedence relationships, etc., are known a priori. The advantages of static scheduling are several. First, no scheduling cost is incurred on the real-time system during run-time. Second, transaction deadlines are guaranteed before the system starts, hence overload situations due to periodic transactions do not arise. Third, high utilization of system resources can be achieved. Finding a feasible or optimal schedule for a certain scheduling problem is most often an NP-complete or at least an NP-hard problem. By performing the search for a feasible solution pre-runtime, virtually, unlimited computer time can be devoted to the search until a feasible schedule is found since minimizing computer time is not a major concern. However, static scheduling has its disadvantages. It requires a priori knowledge of arrival times and execution times that can be hard to attain. Moreover, schedules are subject to change in dynamic environments, but static scheduling offers limited flexibility at

runtime. Static approaches include, for example, cyclic-executive, rate-monotonic scheduling [LL73, SKG91, KRP+93], and deadline-monotonic scheduling [ABR+93, ABW93].

One factor of commonality of static approaches is the set of assumptions that (i) the workload mainly consists of periodic transactions, and that (ii) critical transactions must be periodic. Two approaches have been suggested in order to handle non-periodic transactions. These are the deferrable server [LSS87] and the sporadic server [SSL89]. The server is a periodic process serving aperiodic requests from the environment. This transforms the scheduling problem from scheduling both periodic and non-periodic transactions into scheduling only periodic transactions. However, the transformation does not come cheap. The underlying assumption for these approaches is that non-periodic transactions are of less importance/criticality than the periodic transactions. This implies that, in the presence of an overload situation, non-periodic transactions are dropped and only the periodic transactions meet their deadlines since these are already guaranteed statically. Although this assumption may be valid in some environments and situations, there are situations where periodic transactions are less critical than the non-periodic transactions. For example, assume an environment where the real-time system periodically monitors a set of sensors in a nuclear power plant. In case of an emergency, the system is responsible for responding to external events that trigger aperiodic transactions, initiating damage-control-systems, which is a critical operation. If an overload arises, it is better to drop transactions monitoring the sensors, and instead rely on historical data and give priority to non-periodic transactions.

With *dynamic scheduling*, also known as run-time scheduling, scheduling decisions are made at run-time. The obvious advantage of this class of scheduling algorithms is their ability to adapt to a changing environment. The premises under which dynamic scheduling is adopted are the following. First, the arrival times of the transactions are not known a priori. Second, due to the foregoing premise, overloads may arise which must be resolved. Third, only a limited time can be devoted to find a feasible schedule, which implies that most scheduling algorithms in this class

rely upon heuristics limiting the search space as opposed to exhaustive search-based algorithms. Further, due to the significant run-time cost, the cost must be considered in the scheduling model of the real-time system.

However, as real-time systems increase in complexity, and the environment characteristics evolve from being 'static and fully predictable' to being 'dynamic and predictable', pure static scheduling is not the answer. This motivates research efforts extending pure static scheduling approaches in order to gain a degree of dynamic and adaptable behavior.

In the literature, off-line scheduling and on-line scheduling are often equated to static and dynamic scheduling but, as pointed out by Stankovic et al. [SSDNB95], this is wrong. Analysis is different from scheduling, and off-line analysis should always be applied when building real-time systems, regardless whether the real-time system uses a static or dynamic run-time scheduling algorithm. A dynamic scheduling algorithm is designed to work under certain conditions. Hence, given that some knowledge exists about the dynamic environment in which the real-time system is supposed to work in, then the feasibility and suitability of a scheduling algorithm for an environment can be analyzed.

2.4 The Role of Active RTDBs in Real-Time Systems

When designing real-time systems, mainly two approaches have been used: event-triggered and time-triggered real-time systems. Event-triggered systems react to external events directly and immediately as opposed to time-triggered systems which react to external events at pre-specified instants [KV93]. Event triggered systems are prone to overloads and event showers. Moreover, static scheduling is not applicable due to no a priori knowledge on when events will occur. Dynamic scheduling is necessary, and often preemption is allowed [KV93]. In time-triggered systems, schedules are computed at pre-runtime and are not changed during execution. Characteristically, time-triggered systems are not subject to

overloads and event showers.

As mentioned earlier, real-time systems are inherently reactive, and respond to external events in the physical environment. Events are signaled to the real-time system which, based on the event type, determines what the next step should be. Although real-time systems have possessed reactive behavior, to the best of our knowledge no general methods or techniques for uniform specification of reactive behavior have been developed in the real-time research community.

The active database research community has developed ECA rules which provide a uniform and powerful model of reactive behavior. The model enables the representation of arrival and execution of transactions related to an event occurrence in a controlled environment, which is the typical scenario for many real-time systems. However, active database systems do not explicitly address the incorporation of time constraints. Moreover, active database systems introduce new sources of unpredictability and timeliness issues that must be considered when executing triggered actions. Even though timeouts and temporal events can be represented by rules which are triggered upon the occurrence of a temporal event, there are no mechanisms nor guarantees for timely execution of the triggered action.

Overall, it is not hard to see commonalities between the methodologies used when designing and building real-time systems and active database systems. Moreover, it is easy to see how active database systems can fit into the system model of event-triggered real-time systems.

For more material about active real-time database systems, see [CBB+89, BH95b, AH98, HB98, HA99].

2.5 Active RTDB Preliminaries

With conventional database technology, data is stored in databases that are passive, which only allows us to detect situations or changes to the database by polling the database. Polling is performed by an application, which regularly issues queries to the database and then examines the

results. There are several disadvantages with this. First, given that there is an event or condition of the database which is of global interest, i.e., several applications are interested in the event or condition, then each application is required to detect this by issuing its own queries. This implies a significant overhead because of the duplicated work that is carried out, and further, the time of detection will differ. Second, if short detection times are required, the polling frequency must be high, resulting in a significant workload on the system.

An active database system is a system that monitors the database or the environment in order to react upon state changes in the database or the environment.

The proposed notion of modeling the reactive behavior is to specify rules. Most approaches have adopted the notion of *event-condition-action rules (ECA rules)* [CBB+89].

ON event E
IF condition C
DO action A

The semantics of an ECA rule are: when event E occurs, evaluate the condition C (boolean expression or a method invocation), and if the condition is satisfied then initiate the execution of action A (a database operation or an application program).

Events in active databases are considered to be instantaneous and atomic, i.e. either they happen or they do not. Further, events are classified into primitive events and composite events, where the former is the basic element, i.e., the single constituent of which composite events can be formed by relating a set of primitive events with event operators. Primitive events are normally decomposed into different types of events: database events, temporal events, and transaction events [CBB+89]. A temporal event occurs when a specified point in time is reached. Transaction events occur at begin, commit, and abort of transactions.

While active databases use the ECA concept as a building block for specifying reactive behavior, active database systems do not explicitly

consider the time constraints of transactions, hence even though the ECA rules can be used for specifying timeouts, active database systems do not guarantee the timeliness of the triggered action(s). In HiPAC [CBB+89] it was proposed that a time constraint could be attached to the action part of a rule. This implies a change in semantics as pointed out by [Ram93], where it is identified that traditional ECA rules cannot express the following semantics:

ON event E
IF condition C
DO <COMPLETE> action A <WITHIN t seconds>

The semantics of the above rule action are to *complete* action A within t seconds. In general, the time constraint can refer to (i) the time of event occurrence, or (ii) the time of event detection. The former refers to the actual time when the event was generated, whereas the latter refers to the time when the event was detected by the system. For example, an event manager is most likely to spend a non-trivial amount of time when detecting a composite event [GBLR96].

Depending on whether a transaction deadline is relative to the time of event occurrence or event detection, the importance of the triggered transactions has a dramatic impact on the requirements on the real-time system. If the deadline is relative to the event occurrence, time-cognizant mechanisms for event handling transaction triggering must be catered for explicitly in the system. In contrast, when deadlines are relative to event detection, no upper bound can be obtained for the time between event occurrence and detection. However, real-time systems are inherently reactive and have to respond to external events within a upper bound, implying that transaction deadlines are relative to the time of occurrence (for the event that triggered the transactions). Hence, to guarantee that triggered transaction deadlines are met, the transactions should be triggered before the latest start time of the transaction, with enough time to perform scheduling operations and meet the deadlines of the triggered transactions.

When incorporating reactive behavior in real-time database systems by using techniques and methods developed within the active database community (or vice versa), several incompatibilities and sources of unpredictability can be identified. In the next sections we identify incompatibilities and discuss how these can be addressed in real-time applications.

2.5.1 Components

The reactive mechanisms imply that new functions and services must be provided by the database system. Active database systems include the following additional components (in contrast to conventional database systems): event detector, rule manager, and condition evaluator.

The event detector signals the rule manager upon the detection of an event. Further, the rule manager initiates the condition evaluation of those rules of which the event is a constituent. If the rule condition is satisfied, the rule manager initiates the execution of the action. In order to carry out this task, the rule manager must interact with the transaction manager due to the coupling between the events, conditions, and actions.

2.5.2 Execution Model: Coupling Modes

Actions are carried out during the execution phase. As transactions are executed, new events may be generated which may cause cascaded rule firing and, hence, impose additional workload on the system. Unrestricted cascading of rule firings causes system overload. Hence, cascading must either be bounded with respect to execution time, restricted or prevented. The core issue is how rule execution should be executed with respect to the triggering transactions. Some of the coupling modes are not appropriate for real-time purposes.

When studying the set of coupling modes for event-condition and condition-action, the following combinations of modes are possible in active database systems: (1) *immediate-immediate*, (2) *immediate-deferred*, (3) *immediate-detached*, (4) *deferred-deferred*, (5) *deferred-detached*, (6)

detached-immediate and (7) *detached-detached*. Predictability can be enforced by restricting the set of coupling modes to those not affecting the execution of the triggering transactions.[1]

Event-Condition Coupling

The coupling modes used for event-condition coupling are a source of unpredictability. The real-time scheduler has guaranteed, based on the worst-case execution time of the triggering transaction, that the deadline of the triggering transaction will be met. Given that the triggering transaction generates new events, and that immediate or deferred coupling is used, this implies that the event detection and condition evaluation cost are charged to the triggering transaction. This is due to the necessary blocking of the currently executing transaction. Hence, in order not to override the allocated worst-case execution time, the time consumed on event detection and condition evaluation must be controlled. If this is not done, the coupling combinations 1-5 cannot be adopted in real-time environments. Hence, this also implies that rules are reduced to detached event-condition coupling.

For detached event-condition coupling, condition evaluation is performed in a separate transaction which is scheduled in the same way as any other transaction. The detached coupling mode does not jeopardize the timeliness of the guaranteed transactions or the predictability of the system. However, an additional workload is imposed on the system that must be handled by the real-time scheduler.

Condition-Action Coupling

Given that condition evaluation in the former phase passes the test, i.e., an action should be executed, immediate and deferred actions are then executed as sub-transactions on behalf of the triggering transaction. Actually, similar reasoning as for the immediate and deferred

[1]Detached coupling may be either dependent or independent, but will not affect the list for analysis.

event-condition coupling can be applied. As can be seen, immediate and deferred condition-action coupling may jeopardize the timeliness of the already guaranteed triggering transaction. If the number of cascaded triggered sub-transactions cannot be bounded, and thereby the worst-case execution time of the triggering transaction, then immediate and deferred condition-action coupling have limited applicability in real-time systems. As pointed out by Branding et al. [BB95], the execution time of the transaction is prolonged in proportion to the number of rules triggered in immediate and deferred mode. This may cause blocking delays of other transactions that arrived before the rules were triggered.

As detached condition-action coupling is performed in separate transactions, if event detection and condition evaluation is predictable, then the temporal behavior of the triggering transaction is not affected, i.e., neither predictability nor timeliness is jeopardized. Going back to the list of coupling modes, this means that the coupling combinations 3, 5, 6, and 7 can be adopted in a real-time environment.

Additional types of dependent detached actions have been suggested by Branding et al. [BBKZ94]: causally independent, and parallel, sequential detached and exclusive with causal dependency.

- detached coupling and causally independent – triggered transactions are independent, i.e., no execution constraints are imposed on the triggered transactions;
- detached coupling – sequential and causally dependent, i.e., triggered transaction(s) may not start before the triggering transaction has committed (abort of the triggering transaction will imply abortion of the triggered transaction);
- detached coupling – exclusive and parallel causally dependent, i.e., a transaction τ_j having a parallel and causal dependency on a transaction τ_i, may execute in parallel to transaction τ_i but can only commit if τ_i aborts;
- detached coupling – exclusive and sequential causally dependent, i.e., a transaction τ_j having a sequential and causal dependency on

a transaction τ_i may only execute and commit after the abortion of τ_i.

2.5.3 Event Detection

In real-time applications, the system must respond to a stimulus within an upper bound, where the response is performing a computation and reporting the result. Mapping this to the ECA model, actions are triggered by event occurrences.

Given this, it is interesting to investigate the origin of the time constraints. Time constraints are often assigned to actions, where the time constraints are relative to the event occurrences, i.e., the time association between event occurrences and actions implies that the time constraints of the actions are inherited by the events. By studying the temporal scope of the events, three types can be seen [Ram95, Ram96]: maximum delay of events, minimum delay between events (also known as minimum inter-arrival time), and duration of an event. Moreover, events can be input or stimulus events; output or response events; and internal or invisible events (external events outside the (sub-) system) [Ram95]. In other words, events can be classified according to whether they are in- or outgoing events with respect to the system, and whether the system has awareness of the events or not.

ECA rules provide a good model for specifying the reactive behavior and enforcing constraints, and for triggering actions upon event occurrences. The ECA model, at least in its basic form, does not provide mechanisms for specifying time constraints or guarantee that time constraints are enforced. Timeliness in this case is no longer only a matter of transaction scheduling since the time constraints of the actions are determined by the time constraints of the events. Depending on the characteristics of the event, the deadline may be relative to the event occurrence (typical for external events occurring in the physical environment), or it may be relative to the time when the system detected the event. In the latter case, it is more likely that the event is internal. Hence, in order to obtain a notion of guarantee or schedulability, not only must the set of triggered

transactions be considered, but also methods and algorithms that are performed between event detection and transaction triggering must be predictable or time-cognizant. Hence, methods for event detection, rule selection and triggering, and condition evaluation should be predictable.

2.5.4 Rule Triggering

Upon event detection, the appropriate rules should be selected, that is, those rules that should be triggered as a response the event occurrence. Within active object-oriented database systems, techniques can be broadly categorized into the centralized approach, rules indexed by classes and rules associated with specific events. With the centralized approach, all the rules have to be notified to determine which rules that are subjects for evaluation. By indexing the rules by classes efficiency is increased [BH95a]. Neither method can guarantee that no unnecessary rule triggering is performed. It has been suggested that rules should be associated with specific events, and then notify the rules that are specifically interested in that event [BH95a].

2.5.5 Condition Evaluation

Rule conditions in active databases are implemented as either boolean expressions, methods, or database queries. Sophisticated implementations of active databases can apply techniques proposed for query optimization in order to speed up the condition evaluation [GBLR96]

2.6 DeeDS Architecture

DeeDS is a *d*istributed activ*e* real-tim*e* *d*atabase *s*ystem developed by the Distributed Real-Time Systems Research group at the University of Skövde. Our work on scheduling and overload management forms a basis for the DeeDS system, and for this reason an overview of the system is presented.

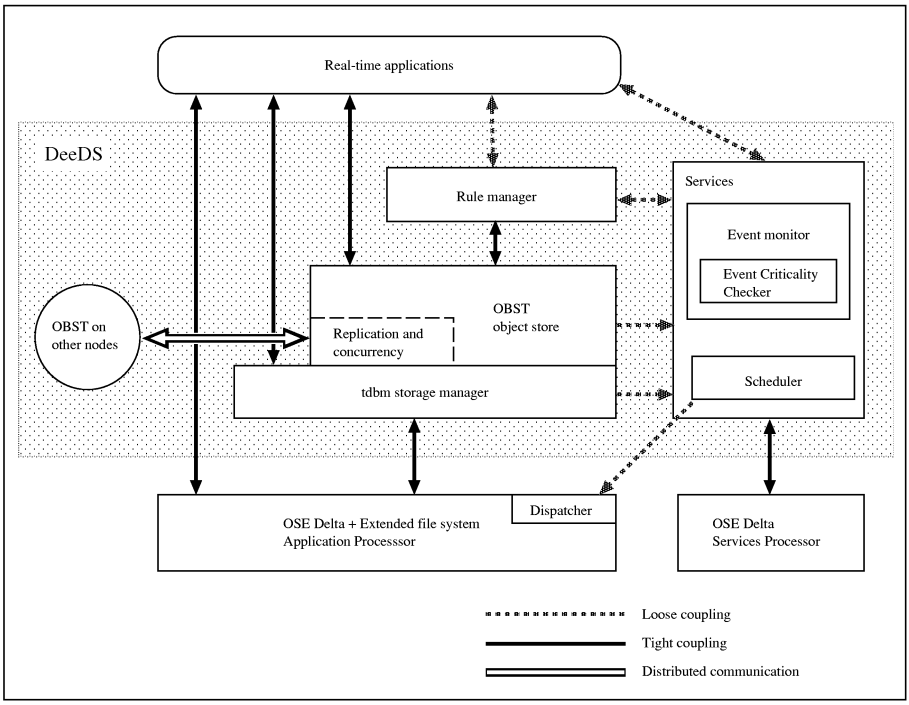


Figure 2.1: The DeeDS architecture

The DeeDS architecture is layered where the layers are closely coupled. The architecture of DeeDS is divided into application-related functions and critical system services.

The application-related functions are decomposed into the rule manager module, the object store (OBST) [CRS+92], the storage manager (tdbm) [BN92], and the operating system OSE Delta. These application-oriented modules are layered. Critical system services consist of the tasks: scheduling, event criticality checking, and event monitoring collapsed into the service module. By replacing the storage manager in OBST with tdbm, nested transactions are supported [Mos85].

The service module works at all layers, where the application-related modules are loosely coupled with the service module.

Main memory residency of the first version of the database is provided by existing main memory resident file systems of OSE Delta.

DeeDS uses dual processing elements, where the service module runs on a dedicated processing element, referred to as service module processor. Its resources have to be shared between the scheduler module and the event monitor. A detailed description of how these services interleave is found in [MHA97].

Application-related functions, as well as application processes, run on the second processing element, referred to as application processor. By separating the system services and the application related functions, the overhead cost model is simplified, especially the cost associated with scheduling decisions. Moreover, the database is main-memory resident.

2.6.1 Rule Manager Module

Rules in DeeDS are be specified in the form of ECA rules extended with time constraints (see [Eri98] for a detailed description of the rule language). The rule manager module provides the following functions: First, *rule storage.* Rules are stored as first class objects. Second, *rule selection.* Events are detected by the event-monitoring module and forwarded to the rule manager. Upon reception of an event instance, the rule-base is searched for rules that are associated with that event, and evaluates that set of rules, if any. Rule conflicts are handled using priorities. Third, *condition evaluation.* The condition part of the ECA rule is evaluated in order to determine whether to execute the action. The condition part is a method invocation. If the condition is true, the action part of the rule is triggered for *action execution.*

2.6.2 Event Monitoring Module

Events are either primitive or composite. Event graphs performing parameter computation are used for detecting composite events. Upon detection, event instances are delivered to interested recipients within a bounded time. The event instances carry information about event type,

time of occurrence, and the scope in which it was generated. The constructors supported in DeeDS are conjunction, disjunction, sequence constructors, and predictable closure constructors in the *recent* and *chronicle* context [CM93, Buc94]. See [Mel] for a detailed description of the event monitor.

If traditional rule detection is adopted, i.e., events are detected in the same order as they occurred, then uncontrolled behavior can result with the timeliness being jeopardized as an effect due to event showers and transient overloads. Berndtsson and Hansson [BH95a] describe a scheme for prioritized event detection and rule handling. Upon the physical event occurrence, event instances are placed in an incoming event queue, which is continuously checked and computed by the event criticality checker, which detects the events. In DeeDS, events either may trigger transactions with hard deadlines (critical) or soft deadlines (non-critical). In order to guarantee that events triggering critical transactions are detected within bounded time, in particular during event showers, events are prioritized. The priority of an event is determined by evaluating the deadline criticality of the action-part of those rules that the event may trigger [BH95a]. In other words, events are categorized into critical and non-critical.

The scheme suggests that events are handled in a strictly prioritized manner, where the priority is determined by the degree of criticality of the most critical action which may be triggered upon that event. Moreover, the tightness of the time constraint of the triggered deadline may also be reflected in the priority assigned. It is suggested that the rule set is analyzed statically in order to determine the criticality of events, which are then parameterized with this information. It is suggested that event showers can be handled by using filtering mechanisms that are sensitive to critical events and thereby can filter out these and present them to the system.

One implication of the approach is that there is a risk that the system will be loaded with critical events. This may happen in systems where it is likely that an event is involved in rules, where the actions vary in criticality, and where the event is likely to be part of at least one rule

with a critical action.

The method can be applied to composite events but with the constraint that the criticality of the composite event cannot be higher than lowest criticality among the constituents of the rule. Hence, the effect is that composite events are likely to be handled as less critical ones.

2.6.3 Replication and Concurrency Control Module

In order to enforce predictability in DeeDS, the following design decisions have been made to isolate unpredictable delays due to disk accesses, network communication and distributed commit processing. First, the database is main memory-resident. Second, the database is (virtually) fully replicated, i.e., each node has local copies of the database objects needed at that node. Third, ASAP (As-Soon-As-Possible) replication is used when propagating updates of the database, implying that temporary inconsistencies may arise. See [Lun97] for a detailed description of how replication is performed.

Chapter 3

Problem Description

This chapter is outlined as follows: In the first section (3.1) we motivate the need for dynamic scheduling algorithms. A detailed description of the nature of the real-time scheduling problem and its complexity is given in section 3.2 and section 3.3 respectively. In section 3.4 we formulate our research goals. Finally, we conclude this chapter by summarizing the scheduling problem (section 3.5).

3.1 Motivation

System correctness of real-time systems is related, in addition to the functional correctness of the result produced, to what extent timeliness of tasks requiring system resources is ensured. In order to enforce explicit time constraints, the real-time system must be designed to handle "real time". As pointed out by Pressman, this makes the design of a real-time computing system the most challenging and complex task that can be undertaken by a software engineer [Pre97]. In a real-time system, the scheduler component has the overall responsibility that timeliness is ensured. Its task is to plan how system resources should be used, outline an execution order that enforces the time constraints, and assign task pri-

orities. For workloads primarily consisting of periodic transactions, the schedulability analysis and priority assignment can be done statically, i.e., prior to system execution. However, static scheduling algorithms have limited or no applicability in application scenarios where the dominant part of the workload is non-periodic, or critical transactions are not periodic, motivating the need for dynamic scheduling algorithms, i.e., transaction priorities are computed and assigned at run-time.

It is generally accepted that missing a single firm or soft deadline "occasionally" is not considered to jeopardize system correctness. This is confirmed by how existing scheduling algorithms for soft and firm real-time systems have been designed and evaluated, attempting to minimize the mean or maximum lateness (soft real-time systems only), or to minimize the number of missed deadlines [Pin95]. Realistically, however, if consecutive instances of a transaction fail to complete before their respective deadlines, then the system will eventually suffer from a failure. This indicates that there are additional constraints, expressing the minimum degree of timeliness for a class of transactions, that should be enforced. For example, such constraints may be expressed as minimum completion ratio for a transaction class (e.g., at least 75% of the firm transaction must complete successfully). Most scheduling algorithms developed for soft and firm real-time systems lack the ability to specify or enforce this type of constraint, which we refer to as a *robustness requirement*. The robustness requirement may specify a temporal interval within which the constraints should be met.

3.2 The Real-Time Scheduling Problem

Real-time systems have a finite set of resources, and hence, have a finite processing capacity. Due to the requirement of guaranteeing temporal behavior with finite processing capacity, the real-time system must be designed to handle peak load situations generated by the environment [KV93]. The peak load is determined both by the transaction workload and by the event load imposed on the system, since event-triggered

systems are prone to event-showers.

The transaction workload, denoted $\mathcal{T}$, is composed of k transaction classes, i.e., $\mathcal{T} = \bigcup_{i=1}^{k} \mathcal{T}_i$. Each transaction class $\mathcal{T}_i$ consists of a set of n_i transactions, $\mathcal{T}_i = \{\tau_{ij} | 1 \leq j \leq n_i\}$, and where transaction classes are disjoint ($\mathcal{T}_i \cap \mathcal{T}_j, i \neq j$). Transaction classes are discriminated by criticality, denoted κ_i, and transactions have hard critical, hard essential, soft, or firm time constraints. The former two are considered critical, while the latter two are non-critical. A transaction τ_i may have one corresponding contingency transaction, denoted $\bar{\tau}_i$, which can be invoked during overloads, replacing the original transaction. A contingency transaction $\bar{\tau}_i$ can, in contrast to the corresponding original transaction τ_i, be characterized as having significantly lesser computational processing requirements with the same or possibly a later deadline. The substitution of an original transaction results in a finite decrease of the utility contributed to the system. Hence, the contingency action should be seen as a last resort that the system could execute in order to handle transient overloads. The utility is represented using value functions [JLT86, Loc86] which express how valuable transactions are to the system as a function of time.

In short, the research problem can best be described as how to dynamically schedule a complex transaction workload and how to gracefully degrade system performance during overloads. The primary criterion is that any schedule must enforce the timeliness of critical transactions requesting resources. Secondly, any schedule should also attempt to ensure that all robustness requirements of non-critical transactions are satisfied. Hence, our primary focus is on the transient overload situations since these are, although hopefully infrequent, the worst threat to the primary criterion.

In this chapter, with the aid of a set of description models, we formulate the details of the scheduling problem that is the focus of our research.

The *transaction model* (section 3.2.1) specifies the transaction types (e.g., transaction importance and criticality) and any inter-transaction dependencies, and constraints concerning the transaction execution order. In

addition, it defines the timing information and resource requirements. Section 3.2.2 specifies the *workload assumptions* i.e., the worst-case workload presented to the real-time system that must be handled. The *reactive model* (section 3.2.3) specifies how reactive behavior is specified. Section 3.2.4 describes the *interrupt and concurrency control assumptions*, i.e., it specifies whether the execution is preemptive or non-preemptive and, in case of preemptive execution, how interleaved execution of transactions is controlled. The *overload model* (section 3.2.5) specifies desirable and acceptable system behavior during transient overloads. Section 3.2.6 describes the *system assumptions*, i.e., the characteristics and assumptions about the system architecture and the hardware model.

3.2.1 Transaction Model

The following temporal attributes, related to the temporal scope constituted by a transaction τ_i, are assured to be known (the corresponding temporal attributes of a contingency transaction $\bar{\tau}_i$ are indicated by a bar, e.g., $\bar{r}_i$):

- *ready time* r_i — the earliest time at which a transaction may start its execution (transactions are ready upon arrival);
- *deadline* d_i — the time at which the transaction execution should be (or must be) complete;
- *deadline tolerance* δ_i — represents tardiness that can be allowed, i.e., when transaction execution must be complete;
- *worst-case execution time* w_i — the maximum execution time of the transaction, independent of the current state of the database and the system;
- *consumed execution time* σ_i — execution time consumed by transaction τ_i
- *remaining execution time* ζ_i — upper bound of remaining execution time of transaction τ_i;

- *value function* $v_i(t)$ — representing the importance and criticality;
- *blocking time* b_i — the maximum amount of blocking time that transaction τ_i may experience due to resources held by lower priority transactions;
- *worst-case execution time (longest critical section)* $wcrit_i$ — the worst-case execution time of the longest critical section of transaction τ_i; and
- *abort time* o_i — the time needed to abort transaction τ_i.

Critical Transactions: Critical transactions are assumed to be either periodic, where the deadline equals the end of the period, or sporadic, where the minimum inter-arrival time is known a priori.

Critical transactions have exactly one contingency transaction $\bar{\tau}_i$. A contingency transaction $\bar{\tau}_i$ may be invoked for execution during overloads, replacing the original transaction τ_i. In other words, contingency transactions have an exclusive commit dependency on the original transaction, that is, either the original transaction commits or the contingency transaction commits.

Non-Critical Transactions: Non-critical transactions, i.e. firm- and soft-deadline transactions, are periodic, sporadic or aperiodic.

Contingency Transactions

Contingency transactions in our model are autonomous executable entities that may be triggered for execution during overloads. Contingency transactions are exclusively and causally detached from the triggering original transaction. The substitution of an original transaction with a contingency transaction will result in a finite decrease of utility contributed to the system. Hence, contingency transactions should normally be seen as a last resort that the system could take to save the situation

(but it could be better than aborting a non-critical transaction in some situations).

DEFINITION 1 *A transaction is said to successfully complete if and only if the original transaction τ_i or the corresponding contingency transaction $\bar{\tau}_i$ completes and meets its deadline.* □

Two types of contingency transactions are used in our model:

Type I Contingency Transactions A contingency transaction of type I performs functions similar to the original transaction but in a cheaper way, with the implication of the result being of reduced quality [LLS+91]. For example, reduction may be in terms of precision, completeness, certainty, or granularity. Typically, quality is traded for timeliness by performing approximate processing. Hence, a type I contingency transaction has significantly smaller computational and processing requirements than the original transaction (i.e., $\bar{w}_i << w_i$). Moreover, its deadline is inherited (possibly relaxed) from the original transaction (i.e., $\bar{d}_i \geq d_i$). In addition, a contingency transaction $\bar{\tau}_i$ has a value function $\bar{v}_i$(t), where $v_i(t) - \bar{v}_i(t)$ $(t \leq d_i)$ quantitatively represents the degree of the quality reduction (or utility loss) due to replacing the original transaction τ_i.

Type II Contingency Transactions A contingency transaction of this type performs counter-actions that eliminate any source of danger or reduce the damage imposed on the system due to the tardiness of the original transaction. Type II contingency transactions are radically different to contingency transactions of type I, since they are taking corrective actions, executing damage control and recovery control. In our system, type II contingency transactions do not necessarily have significantly lesser processing requirements than their original counterparts. The deadlines of type II contingency transactions, relative to the original transactions, are normally extended. As with type I, the value function represents the reduced utility contributed to the system.

3.2.2 Workload Assumptions

Transaction classes are distinguished primarily by their deadline criticality and secondly, their periodicity (periodic, sporadic, and aperiodic). The underlying workload assumptions are:

ASSUMPTION 1 Each transaction τ_i is pre-declared and pre-analyzed with known worst-case execution time w_i. This information is made available to the scheduler and the admission controller as a transaction arrives in the system. □

Estimating the worst-case execution time of transactions in disk-based database systems is a delicate issue due to the unknown blocking times. However, DeeDS is a main-memory resident database and, hence, does not suffer from disk delays or blocking due to disk accesses. Hence, determining the worst-case execution times of transactions is feasible.

ASSUMPTION 2 Each critical transaction has a contingency transaction, which is an autonomous executable entity. Their deadlines and processing requirements are relative to their original transactions, and coupled with causal, sequential and exclusive coupling. □

ASSUMPTION 3 (LOAD HYPOTHESIS) The set of critical transactions is schedulable based on the worst-case execution time of their contingency transactions. □

ASSUMPTION 4 Non-critical transaction classes may have robustness requirements expressed as the minimum class completion ratio, denoted $MCCR_i$, where class completion ratio, denoted CCR_i, is defined as follows:

$$CCR_i = \frac{\#\{\tau_i | success(\tau_j) \wedge \tau_j \in \mathcal{T}_i\}}{\text{total number of transactions of } \mathcal{T}_i \text{ that requested resources}}$$

□

ASSUMPTION 5 The event arrival rate of critical events, i.e., events that are triggering critical transactions, can be handled by the system without jeopardizing the time constraints of the triggered transactions. □

Value Functions

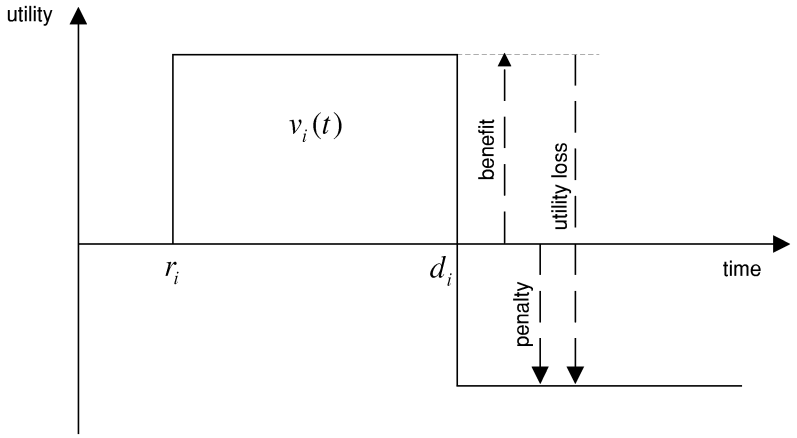


Figure 3.1: Visualization of benefit, penalty, and utility loss

Value (or utility) functions were first suggested by Jensen and Locke [JLT86, Loc86]. The mathematical expressive power of value functions allows universal representation of transaction importance and criticality, and ultimately, enables universal scheduling and handling of transactions, avoiding artificial constructs for handling transactions of different types. Transactions that successfully complete on time contribute a benefit (positive utility) to the system. In contrast, transactions that fail to complete may impose a penalty (negative utility) to the real-time system as a whole, where the penalty is directly related to the criticality of the transaction (see figure 3.1).

In principle, a value function can be seen as, at least, two separate segments. One segment represents the benefit contributed to the system if a transaction finishes within its time constraints, and another segment (or segments) represents the reduced benefit or penalty imposed on the system if it does not successfully complete within the constraints.

Hence, value functions have the advantage that they capture the seman-

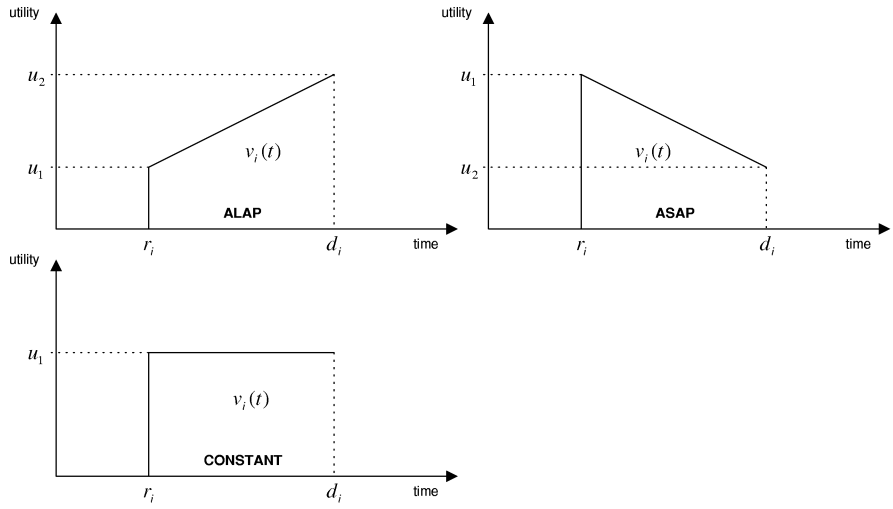


Figure 3.2: Example of typical value functions

tics of the time constraints, e.g., when it is most desirable to execute and complete transactions. Based on the most appropriate time to execute transactions, three basic types of transactions can be distinguished: transactions that should be performed *as soon as possible* (ASAP); transactions that should be performed *as late as possible* (ALAP) (but before the deadline); and transactions not dependent on when they are executed as long as the time constraints are not violated (CONSTANT) (see figure 3.2). This is represented in $v_i^1(t)$, where value functions are piecewise linear. In our work transactions have value functions of the last type, i.e., constant value functions. This is a simplification (for this study) of a more general value function where the utility contributed to the system may change more freely over time.

Biyabani et al. ([BSR88]) suggested how deadline criticality of a task or transaction should be expressed with value functions (see figure 3.3).

For transactions with hard critical, hard essential or firm deadlines, δ_i equals zero, implying that no tardiness is allowed at all. Instead there is no value in continuing the execution of the tardy transaction after the

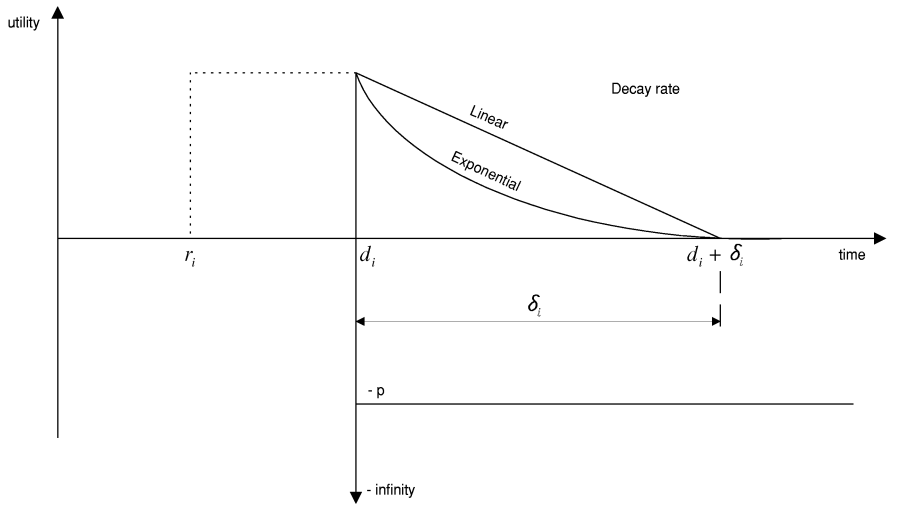


Figure 3.3: Decay rates of value functions after deadline

deadline, and no penalty is imposed on the system.

In our transaction model, we use the following simplified mathematical model to express the value function $v_i(t)$ of a transaction τ_i, yielding the utility contributed to the system when the transaction terminates at time t_i:

$$v_i(t) = \begin{cases} -p_i^1, & t < r_i \ , & 0 \leq p_i^1 \leq \infty \\ v_i^1(t_i), & r_i \leq t_i \leq d_i & \\ v_i^2(t_i), & d_i < t_i \leq d_i + \delta_i & \\ -p_i^2, & t > d_i + \delta_i & 0 \leq p_i^2 \leq \infty \end{cases}$$

We get the following when quantifying the penalty p_i^2 for the different transactions: infinite penalty $(-\infty)$ for hard critical transactions, finite penalty $(-x)$ for hard essential transactions, no penalty (0) for firm and soft transactions. For soft deadline transactions the decay rate of $v_i^2(t)$ is either linear or exponential, becoming zero at time $d_i + \delta_i$. Transactions that have not completed by their deadlines (or within their deadline tolerance) are aborted.

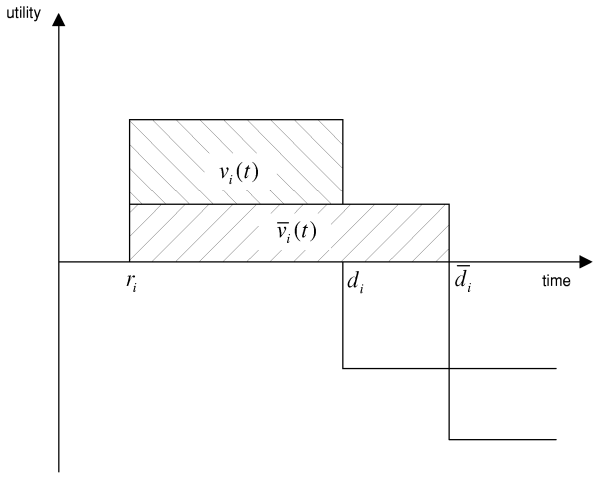


Figure 3.4: Example of how $v_i(t)$ and $\bar{v}_i(t)$ may relate

In figure 3.4 we provide an example of how the value function of a transaction and a value function of a contingency transaction may be related. In this example, we can see that the deadline of the contingency transaction is relaxed (in comparison to its original transaction), i.e., $\bar{d}_i > d_i$. Moreover, $\bar{v}_i(t) - v_i(t)$ is the utility loss, and hence, represents the quality reduction of the result produced by the contingency transaction relative to the original transaction. Further, the penalties are different (often, only the benefit is different).

3.2.3 Reactive Model

Reactive behavior is modeled by ECA rules, where transactions are triggered upon event occurrence, given that a condition is satisfied. A triggered transaction has a coupling relation to the triggering transaction (see section 2.5.2) representing an execution dependency that must be considered by the real-time scheduler.

Coupling modes that introduce unpredictability should be inhibited. An analysis of the basic coupling modes shows that the use of immediate

and deferred transaction coupling in real-time systems introduce unpredictability [Eri98].

This work does not include immediate and deferred coupling between transactions due to the introduced source of unpredictability. In our system, detached coupling, exclusive and sequential causally dependent, is used for coupling contingency transactions to original transactions. Hence, a contingency transaction $\bar{\tau}_i$ may only execute and commit after the abortion of the original transaction τ_i. Furthermore, transactions may trigger new transactions (detached and independent).

3.2.4 Interrupt and Concurrency Control Assumptions

In our scheduling model, transactions are preemptable. Based on the action following the preemption, three types of preemption can be identified: abort of transaction (preemption abort); restart of transaction (preemption restart); or continue the execution from the preemption execution point (preemption resume). The latter type of preemption requires a system mechanism for controlling interleaved transaction execution. In a database environment, this is performed by a concurrency control protocol. We use the optimistic real-time optimistic concurrency control OCCL-SVW [Hua91, HSRT89, HSRT91] in our database model. In our model, transactions are preempted and aborted in cases where a transaction becomes tardy and no longer contributes any benefit to the system. In addition, if transactions are selected for termination in order to release resources with the goal of resolving a pending overload situation, then transactions may be aborted. Transactions subject to data conflicts are restarted.

3.2.5 Overload Model

The handling of transient overloads is of most important concern in this work. As stated earlier, in our system we have a mixture of critical and non-critical time constraints. In order to resolve a transient overload, and minimize the penalty effects of the overload, special care is

needed when managing the transactions. Overloads potentially causing critical deadlines to be missed are the most important to resolve. The consequence of this is that the overload mechanism must be sensitive to what type of overload is about to occur. Hence, in order to resolve a pending transient overload, non-critical transactions may be dropped and transactions having contingency transactions may be replaced.

Our objective is to maximize the total utility given to the system by the transactions submitted to the system as they successfully complete or fail. The total utility is the sum of benefits obtained by successfully completed transactions, and the penalty imposed by tardy transactions. Formally, the scheduling problem can be expressed as (n is the total number of transactions; t_i is the time when τ_i terminates):

$$MAXIMIZE \sum_{i=1}^{n} U_i \text{ where } U_i = \begin{cases} v_i(t_i), \text{if } \tau_i \text{ terminates} \\ \bar{v}_i(t_i), \text{if } \bar{\tau}_i \text{ terminates} \end{cases}$$

3.2.6 System Assumptions

Our real-time database system model incorporates two processing elements, one dedicated to performing scheduling activities (admission control, overload management and scheduling of real-time transactions) and one for executing transactions [AHE+96, MHA97]. Several reasons warrant the use of a dedicated processing element for scheduling and overload management. First, scheduling processors off-load both the scheduling algorithm and other operating system overheads from the application transactions, both for speed and predictability (i.e., external interrupts and operating system overhead do not cause uncertainty in the execution of transactions). Second, the use of special purpose hardware, which is orders of magnitude faster than its alternative software version, enables handling of time granules that are orders of magnitude smaller [BKN+98]. Third, dynamic scheduling has higher run-time costs but offers flexibility and adaptability in contrast to static scheduling. Earlier work on the design and use of a dedicated scheduling co-processor has been carried out in the Spring project [RSZ87, BKN+98]. This system model is consistent with the system design of DeeDS platform. DeeDS

is developed for dual-processor architectures, where one processing element is used for carrying out application transactions, while the second processing element is dedicated for searching for feasible schedules and monitoring events.

The database is main-memory resident and, hence, the blocking of transactions due to disk delays is avoided.

3.3 Complexity of Scheduling Problem

In order to determine the complexity of the scheduling problem at hand, we use a systematic notation introduced by Graham et al. [GLLRK79]. The notation describes a scheduling problem by a triplet $\alpha|\beta|\gamma$, where α describes the machine environment, i.e., the number of processing elements and their characteristics (e.g., same speed, different speeds, etc). The field β describes the nature of the scheduling problem, e.g., characteristics of transactions and constraints. The field γ describes the objective to be optimized. Normally this field contains a single entry. With this model, our scheduling problem can be described as follows (*preemp* refers to preemptive scheduling; *prec* implies there are precedence constraints):

$$1 \mid r_i, preemp, prec \mid \sum U_i \text{ where } U_i = \begin{cases} v_i(t_i) \\ \bar{v}_i(t_i) \end{cases}$$

The termination time t_i is only defined if the transaction τ_i terminates the execution, which also includes tardy transactions that are discarded after their deadline. The precedence constraints are due to the coupling between original transactions and contingency transactions.

We will now show that the scheduling problem is NP-hard. Our proof is based on what in complexity terminology is referred to as problem reduction [Raw91, Pin95, MG87]. Let us formally define this.

Definition 2 (Problem Reduction) *A is* reducible *to B (denoted as $A \propto B$) if and only if there exists a total computable function t such that for any $x, x \in A$ if and only if $t(x) \in B$ [MG87].* □

THEOREM 1 $\propto$ *is transitive.*

Proof. See [MG87]. □

THEOREM 2 *The scheduling problem* $1 \mid r_i, preemp, prec \mid \sum U_i$ *is NP-hard.*

Proof. We prove this by comparing the scheduling problem to a known NP-hard scheduling problem. The following scheduling problem has been shown to be NP-hard [Pin95].

$$1 \mid r_i, preemp \mid \sum W_i \text{ where } W_i = \begin{cases} 1 & \text{if } t_i > d_i \\ 0 & \text{otherwise} \end{cases}$$

The problem $1 \mid r_i, preemp \mid \sum W_i$ is NP-hard, which implies that the problem $1 \mid r_i, preemp, prec \mid \sum W_i$ is NP-hard as well. Hence, we get the following complexity order.

$$1 \mid r_i, preemp \mid \sum W_i \propto 1 \mid r_i, preemp, prec \mid \sum W_i$$

We can see that W_i is a special case of U_i where $v_i(t_i \leq d_i + \delta_i)$ equals 0, and where τ_i has no deadline tolerance ($\delta_i = 0$). Hence, an algorithm for $1 \mid r_i, preemp, prec \mid \sum U_i$ can also be applied to $1 \mid r_j, preemp, prec \mid \sum W_i$. Hence, we get the following complexity order:

$$\begin{aligned} 1 \mid r_i, preemp \mid \sum W_i \quad &\propto 1 \mid r_i, preemp, prec \mid \sum W_i \propto \\ &\propto 1 \mid r_i, preemp, prec \mid \sum U_i \end{aligned}$$

Hence, the scheduling problem $1 \mid r_i, preemp, prec \mid \sum U_i$ is NP-hard.

□

3.4 Objectives

Current state-of-the-art dynamic real-time scheduling algorithms featuring overload tolerance generally adopt only one of the following strategies

for resolving transient overloads: (i) dropping of transactions (discarded upon arrival, or aborted and terminated while executing); (ii) partial execution of transactions; or (iii) execution of recovery actions. The first method is only applicable to non-critical transactions. The second and third methods are particularly applicable to critical transactions, but can advantageously be used for non-critical transactions as well.

At a high level, our research goals are twofold. We want to (i) investigate the effects of multi-class transaction scheduling, where transaction classes are distinguished by their criticality and transactions are handled uniformly upon transaction arrival; and (ii) evaluate the effects of using more sophisticated strategies for overload resolution. More specifically, our goal is to develop a scheduler architecture and a value-driven overload resolution algorithm. In addition, the algorithm should gracefully degrade system performance during transient overloads, and ensure the timeliness of critical transactions by resolving transient overloads using multiple strategies as described in the previous paragraph.

3.5 Summary

In this chapter we have defined and described the characteristics of the scheduling problem we are addressing. For purposes of clarity, we present a short description. The workload consists of multi-class transactions as defined in table 3.1.

Criticality κ_i	*Contingency Transaction* $\bar{\tau}_i$	*Periodicity*	*Overload Resolution Strategy*
critical	required	periodic sporadic	execute $\bar{\tau}_i$
non-critical	optional	periodic sporadic aperiodic	execute $\bar{\tau}_i$ drop τ_i

Table 3.1: Summary of transaction characteristics

The real-time scheduling problem can briefly be described as follows:

- Transactions have critical and/or non-critical time constraints represented by value functions. Critical transactions have contingency transactions that are executed during transient overloads, replacing the original critical transactions.
- Critical transactions are periodic or sporadic with minimum inter-arrival times, while non-critical transactions may be aperiodic. Arrival times of non-periodic transactions are not known a priori.
- Worst-case execution time, periodicity, deadline, deadline criticality, and value functions are known a priori and made available to the scheduler upon arrival of the transaction.
- Original transactions are mutually independent of other original transactions, and transaction execution order is non-constrained. Contingency transactions are detached exclusively coupled to original transactions. Moreover, transactions are preemptable and upon preemption transactions will either be aborted or resumed later.
- Transaction priorities are assigned dynamically, using a dynamic scheduling policy with the ability to handle transient overloads.
- The goal is to gracefully degrade system performance by controllably dropping non-critical transactions and/or replacing transactions with their contingency transaction, as the system becomes overloaded, without jeopardizing the timeliness of critical transactions.

Chapter 4

Framework

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.
- W.L. Bragg

In this chapter we introduce and describe our approach to dynamically resolve transient overloads. In the first section (4.1), we introduce the scheduler architecture and provide a high-level description of its components and their interactions. In section 4.2, the admission control algorithm is described. The transaction scheduler and concurrency control are described in section 4.3. In section 4.4, the heuristic-based overload resolution algorithm OR-ULD and its bias control mechanism OR-ULD/BC are described. The dispatcher is described in section 4.5. In the last section (4.6), implementation issues are discussed.

4.1 Scheduler Architecture

The scheduler architecture consists of an admission controller, a transaction scheduler, an overload resolver, a dispatcher, and transaction queues.

The *admission controller* tests for schedulability of new transactions upon their arrival.

The *transaction scheduler* performs scheduling and concurrency control of admitted transactions by outlining an initial and feasible schedule. The scheduling policy may be any optimal dynamic scheduling policy (in the sense that if there is a feasible schedule, it will find one). In our experiments the scheduling policy is Earliest Deadline First (EDF) [LL73, Der74], blocking is handled by Stack Resource Policy (SRP) [Bak91] and, hence, admissions are based on the schedulability with EDF using SRP.

The *overload resolver* is invoked when a transaction, given its original resource requirements, cannot be admitted to the system. The resolver computes the amount of processing time that needs to be released in order to resolve the transient overload, and initializes a negotiation of the requirements of the admitted transactions and the new transaction.

The *dispatcher* performs dispatching according to the outlined schedule. Two queues are used. The *ready queue* contains the set of admitted but not yet completed transactions. The *rejection queue* contains the rejected and dropped transactions.

4.1.1 Interaction between Components

We will now explain the interaction among the scheduler components (see figure 4.1 where solid lines represent data flow and dashed lines represent control flow). New transactions arriving to the system are placed in an arrival queue. New transactions are then tested for schedulability by the admission controller. The admission controller acts as a transaction filter that denies admission of new transactions in the case of transient overloads, i.e., admission of a new transaction is granted only if it is

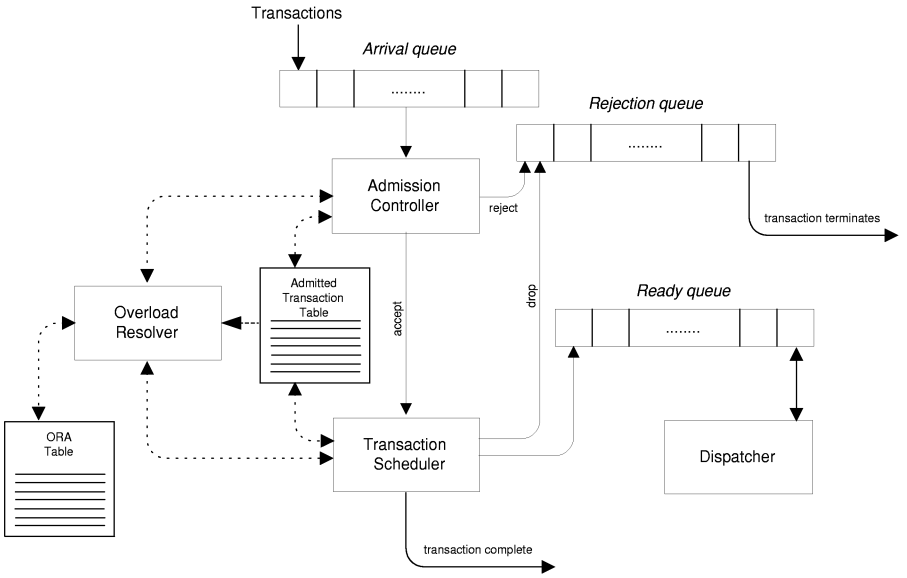


Figure 4.1: Scheduling architecture showing the information flow and component interactions

schedulable with already admitted transactions. Hence, the admission controller guarantees that the transaction scheduler does not become overloaded, i.e., it is able to schedule admitted transactions, although the overall system may be facing a transient overload. If a transaction passes the test, it is admitted to the system and sent to the scheduler. If a transaction fails the test, the admission controller invokes the overload resolver which analyzes earlier resource reservations in comparison to the needs of the newly arrived transaction. That is, the overload resolver determines whether it is worthwhile, given the potential loss of utility due to de-allocation of reserved resources, to accept a new transaction or not. Resources can be released either by dropping a transaction or by replacing the original transaction with a contingency transaction. We call these actions *overload resolution actions (ORAs)*. The overload resolver develops an *overload resolution plan (ORP)*, which consists of a carefully selected set of ORAs. If it is decided that the new transaction

should be admitted, then the overload resolver informs the scheduler, which executes the ORP. If the transaction should be rejected, then the overload resolver notifies the admission controller to reject it. Accepted transactions are assigned priorities by the scheduler and then placed in the ready queue. Transactions that are rejected or dropped are placed in the rejection queue.

Information about admitted transactions is stored in the admitted transaction table (ATT). The workload information in ATT is accessed by the admission controller, the scheduler, and the overload resolver (updates are made by the admission controller and the scheduler only).

To determine which overload resolution actions to select, our approach is to compute the utility loss of performing a certain overload resolution action in contrast to the amount of resources it will release. Hence, the selection of which overload resolution actions to execute, if any, is based on *utility loss density.*

4.2 Admission Control Algorithm

The primary task of the admission controller is to perform a schedulability test of new transactions upon their arrival, determining whether a new transaction should be admitted for execution or not. In our case, using EDF and SRP, the admission controller is pessimistic in the sense that it performs admission based on the worst-case execution times and does not take any chances. Processing time is initially reserved once transactions are admitted. Hence, transactions that are admitted are given a prognosis that they will get their desired level of resources. Resource reservations are scrutinized only in the case of a transient overload. In an overload scenario, given the load hypothesis, this implies that all non-critical transactions must be terminated, and transactions must be replaced with their contingency transactions.

Let $\mathcal{T}_{\mathcal{A}}$ denote the ordered set of admitted transactions as follows:

$$\mathcal{T}_{\mathcal{A}} = \{\tau_i | 1 \leq i \leq n\}, \text{ where } d_{i-1} \leq d_i \text{ for } 1 < i \leq n$$

Moreover, let τ_η represent a new transaction that is tested for admission. Hence, the schedulability test is performed on the unified set, denoted $\mathcal{T}$, of admitted transactions and τ_η, i.e.: $\mathcal{T} = \mathcal{T}_\mathcal{A} \cup \{\tau_\eta\}$.

A critical section is nontrivial if it involves a resource that can cause blocking. With SRP, the blocking time b_i is given by the maximum worst-case execution time of the longest nontrivial critical section of every transaction τ_k such that $d_i < d_k$ $(i \neq k)$. This maximum includes the worst-case execution time of all the critical sections of other transactions that might subject τ_i to priority inversion.

Assuming that the system is at time t_0 before which no transactions are requested, the following condition should be satisfied in order to guarantee the schedulability of a set of periodic and aperiodic transactions using SRP with EDF [Bak91] (deadline d_i is relative to t_0):

THEOREM 3 (BAKER, 1991) *A set of n (periodic and aperiodic) transactions is schedulable by EDF scheduling with SRP semaphore locking if*

$$\forall k = 1, ..., n \left[\left(\sum_{i=1}^{k} \frac{w_i}{d_i} \right) + \frac{b_k}{d_k} \leq 1.0 \right]$$

Proof. See [Bak91, p. 83]. □

Assume that the system has executed for some time and that the currently executing transaction at time t_0 is not in a critical section. The following condition should be satisfied in order to guarantee the schedulability of a set of periodic and aperiodic transactions using SRP with EDF.

COROLLARY 3 *A set of n (periodic and aperiodic) transactions is schedulable by EDF scheduling with SRP semaphore locking if*

$$\forall k = 1, ..., n \left[\left(\sum_{i=1}^{k} \frac{\zeta_i}{d_i} \right) + \frac{b_k}{d_k} \leq 1.0 \right]$$

Proof. Let the subset $\mathcal{T}'$ denote the set of transactions in $\mathcal{T}_\mathcal{A}$ that have started to execute but not yet finished. Then each transaction τ_i in $\mathcal{T}'$

has consumed σ_i time units. Hence, the worst-case execution time for τ_i in order to complete is $\zeta_i = w_i - \sigma_i$. Consider $\tau_i \in \mathcal{T}'$ as newly started transactions with worst-case execution time ζ_i and let $\zeta_i = w_i$ for transactions that have not yet started to execute. □

The pseudo-code for the admission control algorithm testing for schedulability of a transaction workload $\mathcal{T}$ is outlined in figure 4.2.

4.2.1 Description of Stack Resource Policy (SRP)

Baker [Bak91] introduced Stack Resource Policy (SRP) as an improvement to Priority Ceiling Protocol (PCP) [SRL87, SRL90] for handling blocking. SRP has several advantages. The implementation cost and the implementation complexity of SRP are smaller in comparison to dynamic PCP [CL90, CL91]. Moreover, SRP prevents deadlocks and multiple priority inversions in order to bound priority inversion.

Each transaction τ_i has a *preemption level* $\pi(\tau_i)$ (the property of preemption level is that a transaction τ_i is only allowed to preempt another transaction τ_j if $\pi(\tau_i) > \pi(\tau_j)$). In our system preemption levels are ordered reversely with respect to the order of the deadlines, i.e.:

$$\forall i \forall j (\pi(\tau_i) < \pi(\tau_j) \equiv d_i > d_j), i \neq j$$

SRP uses two notions: (current) resource ceiling and (current) system-wide ceiling [Bak91]. The *current resource ceiling* of resource R_i, denoted $\lceil R_i \rceil$, is defined as the maximum preemption level of all the transactions that may be blocked directly by a resource. At any point in time, let the *current system-wide ceiling*, denoted Π, equal the maximum of all the current resource ceilings ($\Pi = max\{\lceil R_i \rceil | i = 1, ..., l\}$ where l is the number of resources).

SRP requires that the following conditions are satisfied [Bak91].

C1 To prevent deadlock, a transaction should not be permitted to start until the resources currently available are sufficient to meet the maximum requirements of the transaction.

```
Admission Controller(T_A, τ_η)
{
    T := T_A ∪ {τ_η}
    for (each τ_k ∈ T) {
        for (each τ_i ∈ T, i ≤ k)
            sum := sum + ζ_i/d_i
        if ((sum + b_k/d_k) > 1.0) then {
            call overload resolver (τ_η)
            return
        }
    }
    admit τ_η
}
```

Figure 4.2: Admission control algorithm

C2 To prevent multiple priority inversion, a transaction should not be permitted to start until the resources currently available are sufficient to meet the maximum requirement of any single transaction that might preempt it.

SRP requires that ceilings must be related to priorities and preemption levels as follows [Bak91, p. 78]:

C3 If transaction τ is currently executing (or can preempt the currently executing transaction) and may request an allocation of resource R that would be blocked by the another transaction currently holding resource R, then $\pi(\tau) \leq \lceil R \rceil$.

Given that condition C3 is satisfied, the following can be established.

THEOREM 4 (BAKER, 1991) *If no transaction τ is permitted to start until $\pi(\tau) > \lceil R \rceil$ for all resources R that the transaction requests, then (i) no transaction can be blocked after it starts, (ii) there can be no deadlock, and (iii) no transaction can be blocked for longer than the duration of one outermost critical section of a lower priority transaction;*

Proof. See [Bak91, p. 79]. □

Using SRP, each transaction is blocked from starting its execution at the time it attempts to preempt if the transaction needs a currently unavailable resource. A transaction τ_i with the earliest deadline is allowed to preempt a currently executing transaction only if its preemption level is greater than the current system-wide ceiling ($\pi(\tau_i) > \Pi$). Once a transaction has started its execution, the transaction will not be blocked since all its resource requests are granted.

In SRP ceilings can be computed off-line and stored in a table. In addition, the locking operations *acquire* and *release* are simpler because they cannot block, i.e., they do not require any blocking test or a context switch [Bak91, p. 96].

Early blocking also reduces the number of unnecessary context switches. Baker showed that the upper bound on the number of context switches caused by a request with SRP is half of the maximum for PCP [Bak91, p. 93].

For a more detailed description of SRP the reader is recommended to read [Bak91, But97, SSRB98].

4.3 Transaction Scheduler and Concurrency Controller

Admitted transactions are assigned a priority according to the EDF priority assignment policy [LL73, Der74]. By exploiting admission control, only schedulable transactions will be admitted, which allows the scheduler to always find a feasible schedule.

The concurrency control scheme adopted is OCCL-SVW (optimistic concurrency control using locking and serial validation write) [Hua91, HSRT91, HSRT89]. Our choice was based on previous performance studies that showed that OCCL-SVW performs better than two-phase locking. However, our proposed framework and overload resolution strategy is independent of OCCL-SVW and can be combined with other concurrency control schemes. Transaction execution using the original optimistic concurrency control has three phases [KR81]. In the *read phase* the necessary data objects are read from the database and then database operations are performed on local copies. In the *validation phase* a check is made as to whether the transaction updates may be in conflict with other transactions. If the validation is successful, the updates are written to the database in the *write phase*.

The following two conditions must be satisfied [Hua91, p. 123], given that transaction τ_i is serialized before transaction τ_j, then

C4 the writes of τ_i should not affect the read phase of τ_j; and

C5 the writes of τ_i should not overwrite the writes of τ_j

Validation can be performed either by backward validation or forward validation. *Backward validation* implies that validation is performed only on committed transactions, and the validating transaction will be aborted if a conflict is detected. *Forward validation* implies that validation is performed on active transactions which in turn means that all committed transactions will be free of conflicts with present (and future) active transactions. Conflicts are here resolved by aborting the validating transaction or by aborting conflicting active transactions in the read phase [HSRT91, Har84]. As pointed out by Huang, in real-time database systems, forward validation is preferable since it provides flexibility for conflict resolution.

We now describe the properties of OCCL-SVW which uses forward validation. OCCL-SVW is deadlock-free by abortion and enforces the conditions C3 and C4. Each transaction τ_i maintains it own *read set*, denoted $\mathcal{R}_i$ and its own *write set*, denoted $\mathcal{W}_i$. The basic OCCL uses two types

of "lock" modes: *read-phase lock* (R-lock) and *validation-phase lock* (V-lock). R-locks are set in the read-phase, and V-locks are set in the validation phase only (on objects that have been updated). An R-lock is incompatible with a V-lock, and a V-lock is incompatible with another V-lock (but R-locks are compatible with each other). If a transaction attempts to set an R-lock on an object already holding a V-lock, then the transaction is forced to wait until the lock can be held. If a transaction attempts to set a V-lock on an object that is already locked by an R-lock, then conflict resolution is invoked [Hua91]. However, explicit V-locks are only necessary when validation and writing of updates are performed in different critical sections (e.g, in OCCL-PVW). Since OCCL-SVW combines validation and writing of updates into one critical section explicit V-locks are not necessary.

Conflicts that are detected during the validation phase are resolved according to the *priority abort* policy [Hua91, p. 131]. The priority abort policy implies that normally conflicting transactions are aborted, but the validating transaction is aborted if its priority is less than that of all the conflicting transactions. The policy considers transaction priority, but still favors the validating transaction, reducing resource wastage due to aborted transactions [Hua91].

The pseudo-code for the OCCL-SVW protocol (using priority abort) is shown in figure 4.3. The symbols '<' and '>' represent the beginning and the end of a critical section.

Performance studies show that OCCL-SVW performs better than the pessimistic two-phase locking scheme when data contention is low; when data contention is high two-phase locking performs better. For more details about the OCCL-SVW protocol, see [Hua91, HSRT91, HSRT89].

4.4 The OR-ULD Overload Management Algorithm

We now introduce the OR-ULD algorithm (Overload Resolution using Utility Loss Density). First, we define temporal intervals necessary for

OCCL-SVW Read Phase (τ):

```
< for (each object in R(τ))
      set R-lock(obj, τ)
>
```

OCCL-SVW Validation and Write Phase (τ):

```
  ValidFlag:=TRUE
< release R-locks held by τ
  for (each object in W(τ)) {
      if (R-locked(obj))
          then {
              ValidFlag:=FALSE
              exit loop
          }
  }
  if (ValidFlag)
      then execute write phase
      else call real-time conflict resolver
>
```

Figure 4.3: OCCL-SVW phases (priority abort): read, validation and write

the OR-ULD algorithm (section 4.4.1). A detailed description of the OR-ULD algorithm and its bias control mechanism (OR-ULD/BC) is then presented in section 4.4.2 and section 4.4.3, respectively. The computational complexity of OR-ULD is analyzed in section 4.4.4.

4.4.1 Overload Intervals

If we are about to resolve an impending transient overload by releasing resources, then given EDF's overload behavior it is important to have an understanding of the time interval over which the transient overload is extending. The reason is, as we will see, that to resolve an overload it is not the case that just any transaction can be dropped or replaced. Consider a set of non-periodic transactions arriving in a system that uses EDF. EDF is known not to handle overloads very well, causing a domino effect of missed deadlines. Let us define two intervals, namely the total overload interval and the critical overload interval.

The workload $\mathcal{T}_\mathcal{A} = \{\tau_1, \ldots, \tau_7\}$ exists in the system and is schedulable by EDF (s_i is the slack time of transaction τ_i, i.e, $s_i = d_i - t_i$). Figure 4.4(a) shows the result of the EDF schedule for $\mathcal{T}_\mathcal{A}$. Correspondingly, figure 4.4(b) shows the result of scheduling $\mathcal{T}$ $(= \mathcal{T}_\mathcal{A} \cup \{\tau_\eta\})$. EDF guarantees that transactions are in order, which determines the insertion point; the marked entries change as a result of the insertion. We can see that the arrival of τ_η results in a transient overload, causing τ_3, τ_4, τ_6 to miss their deadlines. □

Let us formally define TOI and COI (see figure 4.4(c)). Again, consider the situation when the set of transactions $\mathcal{T}_\mathcal{A}$ can be feasibly scheduled with EDF, but where $\mathcal{T} = \mathcal{T}_\mathcal{A} \cup \{\tau_\eta\}$ is not schedulable.

Definition 4 (Total Overload Interval) *The total overload interval (TOI) denotes the total time that the transient overload will last, i.e., from the current time (t_0) until the latest deadline (t_2) of those transactions missing their deadline:*

$$TOI = \big[t_0, t_2\big], \text{ where } t_2 = \max_{\tau_i \in \mathcal{T}_\mathcal{T}} \{d_i\}$$

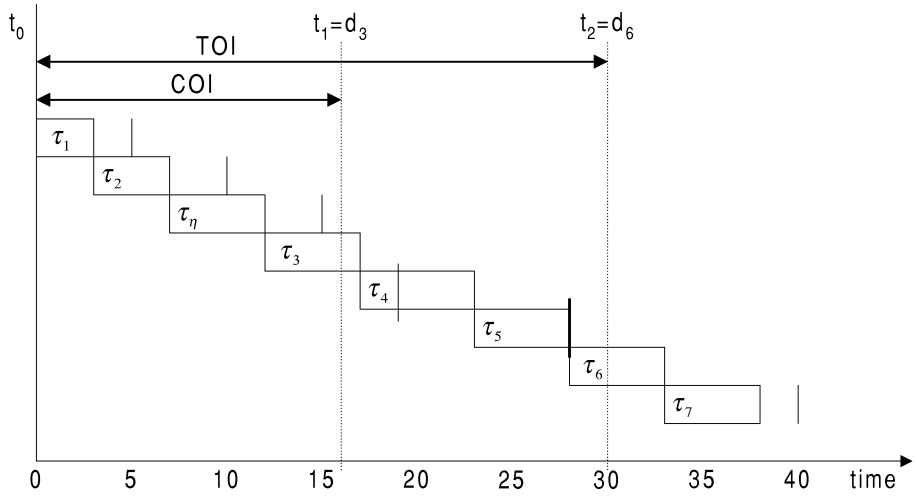
Example 1 (Visualization of Overload Intervals)

τ_i	w_i	d_i	t_i	s_i
τ_1	3	5	3	2
τ_2	4	10	7	3
τ_3	5	16	12	4
τ_4	6	19	18	1
τ_5	5	28	23	5
τ_6	5	30	28	2
τ_7	5	40	33	7

(a) EDF schedule of $\mathcal{T}_\mathcal{A}$

τ_i	w_i	d_i	t_i	s_i
τ_1	3	5	3	2
τ_2	4	10	7	3
τ_η	**5**	**15**	**12**	**3**
τ_3	5	16	**17**	**-1**
τ_4	6	19	**23**	**-4**
τ_5	5	28	**28**	**0**
τ_6	5	30	**33**	**-3**
τ_7	5	40	**38**	**2**

(b) EDF schedule of $\mathcal{T}$



(c) Graphical illustration of EDF schedule of $\mathcal{T}$

Figure 4.4: Example of overload intervals

where $\mathcal{T}_\mathcal{T}$ is the set of tardy transactions[1] *in $\mathcal{T}$, i.e.,*

$$\mathcal{T}_\mathcal{T} = \{\tau_i | \tau_i \in \mathcal{T} \wedge t_i > d_i)\}$$

□

De-allocation of processing time is done in the interval referred to as the critical overload interval. Informally, the *critical overload interval (COI)* starts once the overload is detected at time t_0 (current time), and ends at the time of the deadline of the first transaction missing its deadline. Note, the critical overload interval is smaller or equal to the total overload interval, and resolving the overload here using the minimum required time is guaranteed to resolve it in the total overload interval. However, releasing the same time outside the critical overload interval will not resolve the transient overload.

Definition 5 (Critical Overload Interval) *The critical overload interval $[t_0, t_1]$ denotes the interval starting at time t_0 when the overload is detected at time t_0, and ends at time t_1 when the first transaction is missing its deadline:*

$$COI = [t_0, t_1], \text{ where } t_1 = \min_{\tau_i \in \mathcal{T}_\mathcal{T}} \{d_i\}$$

□

In our approach, de-allocation of processing time is done in the critical overload interval. If enough processing time can be released in this interval, then the overload will be resolved. In other words, resolving the overload in the critical overload interval is guaranteed to resolve it in the total overload interval, and de-allocating time only outside the critical overload interval will have no effect on resolving the overload.

4.4.2 Overload Resolver

The basic idea behind the overload resolver is to generate an overload resolution plan (ORP) that resolves the impending transient overload by

[1] "Tardy transactions" refer to transactions missing their deadline.

de-allocating time from previously admitted transactions. Thus, in the case of an overload, the resource reservations are scrutinized, and they may be reduced by substituting transactions having contingency transactions, or de-allocated by dropping transactions. Hence, transactions that are admitted are only given a prognosis that they will get their desired level of resources, i.e., a conditional guarantee. Dropping and replacing transactions result in a utility loss relative to the initial expectations (processing time is wasted if the transactions have already started to execute). Hence, it is of interest to minimize the utility loss. Note that a new transaction will eventually provide (when complete) some utility to the system, and this must be contrasted with the utility loss caused by de-allocating sufficient resources in order to admit the new transaction.

More precisely, the resolver

(i) determines COI and TOI, and computes the amount of processing time (Δs) that needs to be de-allocated in COI in order to resolve the overload in TOI;

(ii) generates one or more nearly optimal overload resolution plans (ORPs), where an ORP consists of a set of overload resolution actions (ORAs) that de-allocate resources among admitted transactions; and

(iii) decides whether it is advantageous to carry out one of the ORPs considering the relative utility loss or gain of executing the ORP and accepting the new transaction in comparison to simply rejecting it (non-critical) or substituting it (critical).

From the set of valid ORAs, i.e., ORAs with finite utility loss (and release resources in the critical overload interval), we build an ORP by selecting a subset of ORAs that minimizes the total utility loss while de-allocating the required amount of time to admit the new transaction. ORAs imposing an infinite penalty by dropping critical transactions will not be generated since these are, by definition, not considered valid.

Computing the Time Saved by an ORA

The amount of *saved time*, denoted ξ_i^x, in a critical overload interval as a result of performing a specific ORA (d=drop or r=replace) on an arbitrary transaction τ_i, is computed as follows. When dropping the transaction τ_i, the amount of time de-allocated is:

$$\xi_i^d = \zeta_i - o_i + \psi_i$$

where ζ_i is the *remaining execution time*, o_i expresses the *time for aborting* the transaction, and ψ_i is related to *blocking time*, defined as follows:

$$\psi_i = max(0, wcrit_i - \underset{j}{max}\{b_j | j \neq i\})$$

Since we are using SRP, we know that a high-priority transaction τ_i may be blocked only once by a low-priority transaction. Thus, b_i is the worst-case execution time of all critical sections among low priority transactions for transaction τ_i. If transaction τ_i has the longest critical section, then $wcrit_i$ time units have been reserved for blocking, but if it is decided to drop or replace transaction τ_i, then transaction τ_i will not enter its critical section and, hence, will not block any other transaction, implying that time may be saved. The amount of time saved is then the difference in execution time between the longest critical section of τ_i ($wcrit_i$) and the longest critical section b_j ($j \neq i$).

The amount of time de-allocated when an original transaction τ_i is replaced by its contingency transaction $\bar{\tau}_i$, having the same deadline (i.e., $d_i = \bar{d}_i$) can be computed in a similar way. Here, the execution time of the longest critical section of the contingency transaction $\bar{\tau}_i$ has to be considered, i.e., we get the following:

$$\xi_i^r = \zeta_i - o_i - \bar{w}_i + \psi_i$$

where

$$\psi_i = max\left(0, wcrit_i - max\{\bar{b}_i, b_j | i \neq j\}\right)$$

Time Saved due to Non-Occurring Blockings in an ORP

In the case an ORP is composed of several ORAs, the actual amount of time saved due to non-occurring blockings depends on which transactions are selected to be dropped or replaced and, hence, cannot be finally computed before ORAs are selected. The following example shows why. Consider the set of transactions $\{\tau_i, 1 \leq i \leq n | d_{i-1} < d_i, 1 < i \leq n\}$. Further, assume that τ_k has the longest critical section. This implies that τ_k may block another transaction for $wcrit_k$ time units. Hence, if we decide to drop τ_k we will save a further $wcrit_k - wcrit_j$ time units, where $wcrit_j$ represents the second longest critical section. If we decide to drop or replace both τ_k and τ_j, the time saved will be the difference between $wcrit_k$ and the time of the third longest critical section. However, if we decide to only drop τ_j, no time will be saved with respect to blocking, since any transaction may still be blocked for (at most) $wcrit_k$ time units. In conclusion, at the time that ORAs are generated there is no knowledge of which ORAs will be selected to be part of an ORP, which makes it hard to estimate any additional time saved due to non-occurring blocking. In our model, the amount of time saved by an ORA is initially computed using only the remaining execution time, the time needed to abort a transaction, and possibly, the execution time of the contingency transaction. In other words, ψ is initially considered to be negligible. This computation is determined at the time when ORAs are generated. Computing the time saved due to blocking that will no longer occur is postponed until the actual set of ORAs that have been selected to be part of the ORP is known. Then it is possible to determine the amount of time saved due to non-occurring blocking since we know which transactions are subject to be dropped or replaced and how these transactions relate to other transactions. This simplification reduces the computational complexity of the algorithm.

Computing the Utility Loss Caused by an ORA

The amount of *utility loss*, denoted γ_i^x, as a result of a specific ORA (d or r) on an arbitrary transaction τ_i is computed as follows. When

dropping the transaction, the utility loss is given by the loss of benefit of meeting the deadline and the penalty of missing the deadline d_i (in the case where a transaction has a firm deadline, the utility loss thus equals the loss of benefit only), i.e.:

$$\gamma_i^d = v_i(t_i) - v_i(t_i')$$

where t_i ($t_i \leq d_i$) is the original termination time, and where t_i' ($t_i'' > \bar{d}_i$) is the new effective termination time.

By replacing a transaction τ_i with its contingency transaction $\bar{\tau}_i$, the amount of utility loss is the difference between the benefit contributed by τ_i and the benefit contributed by $\bar{\tau}_i$ ($\bar{\tau}_i \leq \bar{d}_i$), i.e.:

$$\gamma_i^r = v_i(t_i) - \bar{v}_i(\bar{t}_i)$$

Algorithm for Generating the Overload Resolution Plan

OR-ULD attempts to select the best ORAs for saving the required amount of time in the critical overload interval. Selection of ORAs is made by relating the utility loss caused to the amount of resources that are de-allocated by an ORA. OR-ULD computes the *utility loss density* (γ_i^x / ξ_i^x) for each ORA, and lets the set of actions be ordered by their utility loss density. The algorithm consists of the following steps (the pseudo-code for the OR-ULD algorithm is outlined in figure 4.8):

(i) Generate the set of possible ORAs in COI and compute their utility loss density.

(ii) Sort ORAs by utility loss density.

(ii) Iterate through the ORAs in order of decreasing utility loss density, and add them to the ORP until the total amount of saved time by the new ORP (overload resolution plan) is greater or equal to the amount of time required.

In order to provide a better understanding of how the algorithm for selecting ORAs works, and what computations are performed, we present an example.

EXAMPLE 2 Overload Resolution by OR-ULD

Consider a set of admitted transactions $\mathcal{T}_\mathcal{A} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, where τ_1 and τ_2 have critical deadlines and contingency transactions ($\bar{\tau}_1$, $\bar{\tau}_2$), and τ_3, τ_4, and τ_5 have firm deadlines (the characteristics of the transactions are given in figure 4.5(a) on page 70). A new transaction τ_η arrives to the system at time t=30 causing a transient overload. If τ_η is admitted and scheduled, the consequence is that τ_2 and τ_5 will miss their deadlines (figure 4.6(a) shows the EDF schedule for $\mathcal{T}_\mathcal{A}$ and figure 4.6(b) shows the EDF schedule for $\mathcal{T} = \mathcal{T}_\mathcal{A} \cup \{\tau_\eta\}$).

In order to admit transaction τ_η, w_η time units must be available. Since, τ_2 has the smallest slack of transactions having a later deadline than τ_η, τ_2 is the transaction that will be affected the most. Before τ_η arrived to the system, τ_2 had a slack of $s_2 = 20$ time units and, hence, if τ_η is admitted τ_2 will miss its deadline by $\Delta s = 30$ time units ($s_2' = w_\eta - 20$). Therefore, if τ_η were admitted, then 30 time units must be released in the critical overload interval.

OR-ULD goes through the list of possible and valid ORAs, sorted by utility loss density in ascending order (see figure 4.5(b)). The algorithm adds ORAs to the selected set $\mathcal{A}_{\mathcal{ORP}}$ until enough time has been released. Hence, we get $\mathcal{A}_{\mathcal{ORP}} = \{\alpha_3^d\}$. The cost of $\mathcal{A}_{\mathcal{ORP}}$ is 200, which should be compared to the utility contributed by τ_η and the penalty of rejecting it. Although the cost of $\mathcal{A}_{\mathcal{ORP}}$ is higher than the utility contributed by τ_η, admitting τ_η is a better choice since the alternative overload resolution plan, suggesting that τ_η should be rejected, imposes infinite utility loss and, hence, is not valid. If we consider admitting the contingency transaction $\bar{\tau}_\eta$, we need to release ten time units ($\bar{w}_\eta - 20$). As it turns out, in this example, the algorithm will produce an overload resolution plan equal to the former one and, hence, the cost efficiency of admitting $\bar{\tau}_\eta$ is worse. Considering this, the overload resolver will decide to resolve the overload by dropping τ_3 and admitting τ_η. The new EDF schedule is shown in figure 4.7. □

By selecting ORAs in reverse order of utility loss density, we ensure that de-allocated time is cost efficient with respect to other ORAs. Given

τ_i	r_i	w_i	d_i	$v_i(t_i \leq d_i)$	$v_i(t_i > d_i)$
τ_1	0	80	430	350	$-\infty$
$\bar{\tau}_1$	0	30	430	200	$-\infty$
τ_2	30	80	280	350	$-\infty$
$\bar{\tau}_2$	30	40	280	100	$-\infty$
τ_3	50	100	260	200	0
τ_4	100	50	250	400	0
τ_5	110	50	350	300	0
τ_η	120	50	230	180	$-\infty$
$\bar{\tau}_\eta$	120	30	230	100	$-\infty$

(a) Characteristics of transaction workload $\mathcal{T}$

α_i^x	γ_i^x	ξ_i^x	γ_i^x/ξ_i^x	valid	comment
α_3^d	200	45	4.44	yes	
α_2^r	250	45	5.55	yes	
α_5^d	300	50	6.00	no	(remaining time not in COI)
α_4^d	400	50	8.0	yes	
α_1^r	150	15	10	no	(remaining time not in COI)
α_1^d	∞	35	N/A	no	
α_2^d	∞	45	N/A	no	

(b) Overload resolution actions

Figure 4.5: Overload resolution example: Transaction workload and the generated set of ORAs

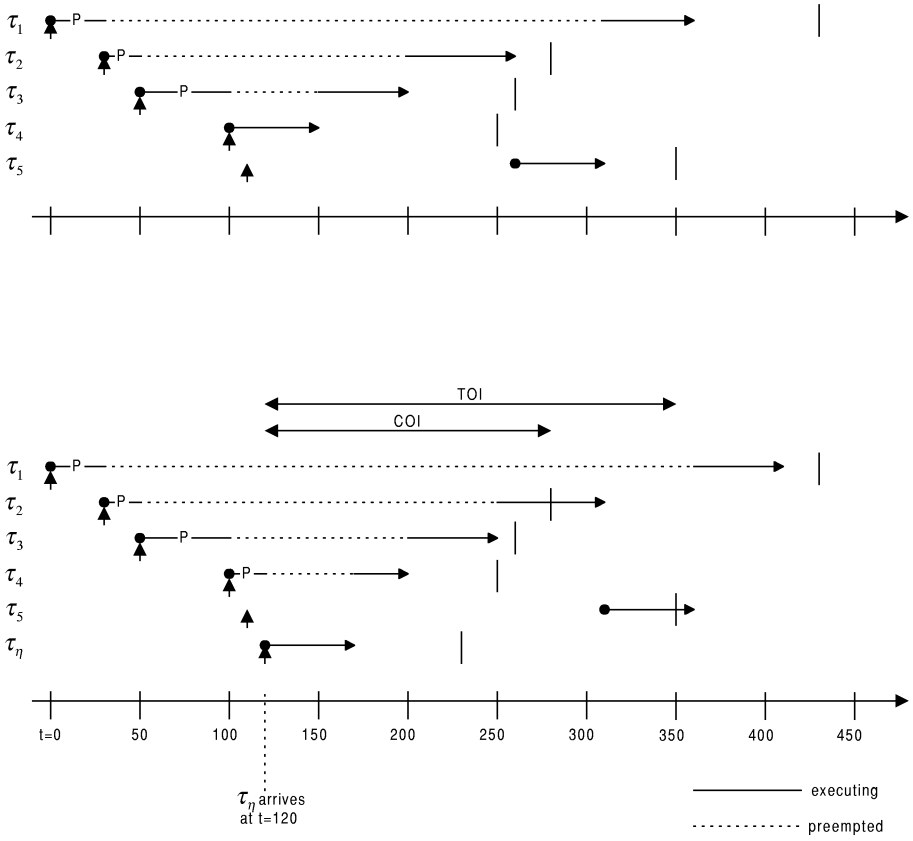


Figure 4.6: An example of overload resolution using OR-ULD: EDF schedule for (a) $\mathcal{T}_\mathcal{A}$ [top] and (b) $\mathcal{T}$ [bottom]

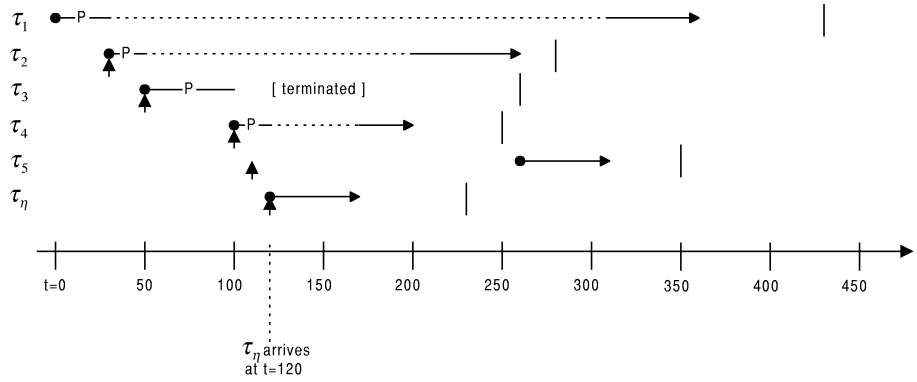


Figure 4.7: EDF schedule after overload resolution

a workload where transactions are similar in length and utility, transactions close to completion will not be selected due to the limited time they would save if dropped. Critical transactions cannot be dropped, only replaced, and the amount of time saved by replacing a transaction is often more limited. Once the remaining execution time of a critical transaction is smaller than the worst-case execution time of its corresponding contingency transaction, replacing the critical transaction is not justified in the general case.[2]

As we have seen in example 2, when an arriving transaction has a contingency transaction there is an opportunity of directly accepting the contingency transaction instead. Admitting the contingency transaction requires less, if any, processing time to be de-allocated than the original transaction, but it also contributes less utility. The amount of time that needs to be released in COI in order to admit τ_i ($\bar{\tau}_i$) is $\Delta s = w_\eta - S$ ($\Delta s = \bar{w}_\eta - S$), where S is defined as follows:

$$S = \min_{\tau_i \in \mathcal{T}_\mathcal{S}} \{s_i\} \text{ where } \mathcal{T}_\mathcal{S} = \{\tau_i | \tau_i \in \mathcal{T}_\mathcal{A} \wedge d_i > d_\eta\}$$

We generate two ORPs and compute the relative cost of implementing them (note that (ii) and (iii) are exclusive): (i) admitting original transaction τ_η; (ii) admitting contingency transaction $\bar{\tau}_i$ (critical transactions only); and (iii) rejecting transaction τ_i (non-critical transactions only). The most cost-effective approach will then be selected, i.e., the one with the best overall utility gain or loss. The overall change in utility, denoted Φ, can be computed as follows, and the ORP with the lowest overall utility loss (or highest utility gain) will then be selected.

$$\begin{aligned}
&\text{admit } \tau_\eta: && \Phi_1 = v_\eta(r_\eta \leq t_\eta \leq d_\eta) - \textbf{OR-ULD}(w_\eta - S) \\
&\text{admit } \bar{\tau}_\eta: && \Phi_2 = \bar{v}_\eta(r_\eta \leq \bar{t}_\eta \leq \bar{d}_\eta) - \textbf{OR-ULD}(\bar{w}_\eta - S) \\
&\text{reject } \tau_\eta: && \Phi_3 = -v_\eta(r_\eta \leq t_\eta \leq d_\eta) + v_\eta(t'_\eta) \text{ where } t'_\eta > d_\eta
\end{aligned}$$

It should be noted that in our approach transactions may be downgraded, or even dropped, by the overload resolver. Once this happens,

[2]The exception is if the deadline of the contingency transaction is later than its original transaction and postponing the execution of the contingency transaction will save time in the critical overload interval.

```
OR-ULD(in Δs, T_A; out sul, A_ORP)
{
    time sst := 0      /* sum of saved time */
    time sul := 0      /* sum of utility loss*/
    set A_ORP := ∅     /* selected set of ORAs */
    set A_ORA          /* generated set of ORAs */

    /* Generate ORAs */
    generateORAs(A_ORA)

    /* Sort ORAs based on utility loss density (ascending order) */
    sortORAs(A_ORA)

    /* Select ORAs based on utility loss density */
    for (each α_i^x ∈ A_ORA) {
        A_ORP := A_ORP ∪ {α_i^x}
        sst := sst + ξ_i^x
        sul := sul + γ_i^x
        if (sst ≥ Δs)
            then return
    }       /* for loop */
    sul := ∞
    A_ORP := ∅
    return
}
```

Figure 4.8: Overload resolution algorithm

a transaction will not be upgraded again. We have considered the cost of frequent invocations of a transaction upgrader as significantly higher than the pay-off. Of course, this may depend on the characteristics of the application considered.

4.4.3 Bias Control (/BC)

Since non-critical transaction classes may have additional requirements regarding their minimum completion ratio (see section 3.2.2), resolving overloads becomes a more delicate issue. Any feasible schedule should, in addition to enforcing time constraints of critical transactions, ensure that the following constraint is not violated:

$$\forall i(\kappa_i = \text{firm} \supset CCR_i \geq MCCR_i)$$

In this situation we can identify two types of overload. The first type of overload (non-critical) is due to resource shortage where not all transactions can meet their deadline. The second type of overload (critical) is due to resource shortage and the lack of a feasible schedule that does not violate robustness requirements. The consequence is that the overload mechanism must be sensitive to the type of overload that is about to occur. The non-critical overload is handled by performing admission control only. Considering the worst-case execution time w_i of each transaction τ_i, we can ensure that the set of admitted transactions is schedulable. For transaction workloads where transactions are non-critical, transactions are normally rejected once it is determined that they cannot be admitted given the current workload. For multi-class transaction workloads, where transactions are of different criticality and have different requirements, overload handling becomes harder. OR-ULD handles this type of transaction workload. The critical overload is handled by the admission controller and OR-ULD in coordination.

The OR-ULD algorithm does not enforce robustness constraints, i.e., the minimum class completion ratio ($MCCR_i$) of transaction class $\mathcal{T}_i$ (see section 3.2.5). Given that utility loss density is the driving force when resolving overload, non-critical transactions with low utility or relatively

long execution times may be discriminated. If there is no constraint on the miss ratio of any transaction class, then this correct and desirable behavior maximizes total utility. However, to ensure robustness of the system, the overload management algorithm must balance resources among transaction classes fairly. For this purpose, a bias control mechanism has been added to OR-ULD. The resulting algorithm is called OR-ULD/BC.

Transaction classification is based on criticality (κ_i) and $MCCR_i$, i.e., transactions within class $\mathcal{T}_i$ have the same criticality and requirement on minimum completion ratio. CCR_i represents the actual completion ratio of transaction class $\mathcal{T}_i$, and, hence, the goal of our scheduling and overload management algorithm is to ensure that the class completion ratio CCR_i never drops below $MCCR_i$. Ultimately, the desirable behavior during the highest acceptable overload (the boundary of the operational envelope) is that the completion ratios of all transaction classes are close to their respective minimum completion ratio requirement (i.e. $\forall i\,(CCR_i \gtrapprox MCCR_i)$). In order to enforce this constraint in a value-driven approach, we add a bias to the utility by, conceptually, increasing the utility gained if the transaction completes as planned.

$$v'_j(t) = v_j(t)\frac{\rho(1.0 - CCR_i)}{(1.0 - MCCR_i)}, \text{ where } \tau_j \in \mathcal{T}_i, MCCR_i < 1.0$$

The bias added to the utility increases as the completion ratio decreases. The variable ρ is a bias parameter.

Increasing the utility with a bias does not affect the schedulability of the admitted transactions; it only affects the decision of which transactions should be admitted. Hence, the increased utility value is only used when the transaction is tested for admission and when the overload resolver is negotiating the reserved resources. The biased value is not used when computing the actual utility contributed to the system.

4.4.4 Computational Complexity of OR-ULD

In order to determine the computational complexity of OR-ULD, we use the "sum rule" of "order notation" [Raw91]. This rule helps us analyze programs that have a sequence of parts with run-times of different orders of magnitude.

THEOREM 5 *The run time of OR-ULD is $O(n \log n)$, where n is the number of transactions.*

Proof. In order to analyze the computational complexity of selecting ORAs with OR-ULD, we must analyze the computational complexity of each step performed by OR-ULD. These are: (i) generate ORAs and compute utility loss density, (ii) sort ORAs by their utility loss density, and then (iii) select a set of ORAs for an ORP.

The first step is to generate all possible ORAs, i.e., ORAs that release time in COI. Assume that $\#\mathcal{T}_{\mathcal{A}}$ (the number of admitted transactions) is n, and that the maximum number of ORAs generated for each admitted transaction is c, where c is constant ($c = 2$ if the ORAs are drop and replace only). Then the maximum number of ORAs that could be generated is $c * n$, and generating $c * n$ ORAs takes $O(n)$ run time (since the time to generate an ORA is $O(1)$).

The second step is to sort ORAs based on their utility loss density. Sorting can be performed by using, for example, heap sort, merge sort, or quicksort (for a description of these algorithms the reader is referred to [Knu98, DLSB82, Wei99]). Heap sort and merge sort run in $O(n \log n)$. The average running time of quicksort is $O(n \log n)$; its worst-case running time is $O(n^2)$, but this can be made exponentially unlikely with little effort [Wei99]. In conclusion, sorting can be done in $O(n \log n)$ time.

In the third step, composing an ORP, the algoritm iterates through the sorted set of ORAs. In the worst case, the entire set of ORAs must be checked, i.e., $c * n$ steps. Note that the algorithm terminates once the time saved by the selected ORAs is greater than or equal to the

required amount of time. If the algorithm iterates through the entire set of ORAs, without being able to release the required time, then OR-ULD terminates since it is not able to resolve the overload, which is contrary to assumptions. The maximum number of selection steps equals the number of ORAs generated ($c * n$), implying that the running time of this step of the OR-ULD algorithm is $O(n)$. Thus, the total number of steps performed by OR-ULD is:

$$O(n) + O(n \log n) + O(n) = O(n \log n)$$

□

4.5 Dispatcher

Admitted transactions are dispatched by the dispatcher residing on the application processor. In the case where a currently executing transaction needs to be preempted due to the arrival of a high-priority transaction, the scheduler informs the dispatcher about this.

When a transaction completes, information regarding any gained (unused) processing time is reported by the dispatcher back to the scheduler.

4.6 Implementation Issues

The OR-ULD and OR-ULD/BC algorithms require the following internal information structures and queues:

- Admitted Transaction Table. This table contains information about admitted transactions ($\mathcal{T}_\mathcal{A}$), arranged in the order they should be executed on the application processor. The admitted transaction table contains, in addition to the identity of the admitted transactions, their remaining execution time (updated by dispatcher), slack, and deadline. The structure is used by the admission controller, the scheduler and the overload resolver.

- Overload Resolution Action Table. The table contains the generated set of ORAs. Note, only valid ORAs can be selected for overload resolution. The table contains the following fields: identity (i) of transaction, type of ORA (drop/reject), utility loss (γ_i^x), time saved (ξ_i^x), and the computed utility loss density (stored for convenience only when sorting). The structure is used by the overload resolver only.

- Ready Queue. This queue contains a list of admitted transactions sorted by execution order.

- Rejection Queue. This queue contains non-tardy transactions that have been rejected or dropped due to resolving a transient overload, but which could be considered for renewed execution in the case where the system becomes underutilized again.

The overload resolver assumes that the remaining execution time of a transaction is known. In real systems, the remaining execution time can be estimated by monitoring the amount of time executed so far (σ_i), and, since the worst-case execution time (w_i) is known, the remaining execution time is given as $\zeta_i = w_i - \sigma_i$. The performance deficiency of this approach is that the difference between the actual remaining execution time and the worst-case time is likely to increase as the transaction gets closer to completing. The only way to avoid this is to monitor actual progress of the transaction and refine w_i at milestones.

Chapter 5

Results

Insight, untested and unsupported, is an insufficient guarantee of truth.
- B. Russel

This chapter presents the results derived from the performance analysis of OR-ULD and OR-ULD/BC. We first present the simulation environment, motivating the use of a simulation model, followed by a description of the the simulator (section 5.1). Section 5.2 describes the simulation experiments. An analysis of the findings regarding the overload behavior of OR-ULD is given in 5.3. Section 5.4 presents the extended analysis incorporating the bias control mechanism (OR-ULD/BC). Finally, section 5.5 presents guidelines for assigning value functions to transactions.

5.1 Simulation Environment

5.1.1 Motivation for Using Simulation

Analytical methods have proven very useful on real-time scheduling problems where complete a priori knowledge about the system and the workload is available. These real-time systems are typically statically scheduled. However, dynamic real-time scheduling problems tend to be more complex to analyze than their static counterparts. Simulation has been used extensively as an empirical analysis tool for performing performance and behavior analyses of scheduling algorithms. This is particularly true for the class of complex dynamic real-time scheduling problems where analytical solutions are not yet available or where analytic methods cannot be applied.

Simulation has been employed in order to evaluate the performance of OR-ULD and OR-ULD/BC, to gain insight about their behavior, and to identify which variables, if any, are important and affect their behavior. Moreover, by changing simulation inputs and observing the resulting outputs, possible dependencies between variables can be identified.

Furthermore, simulation has the advantage that rarely occurring real-life situations can be simulated and analyzed, e.g., transient overloads. Gaining the same knowledge by performing real-life testing is likely to be more time-consuming depending on how often situations and phenomena arise, and it might not be possible at all due to the catastrophic consequences a transient overload may cause. In addition, the stability of the system, and the underlying subsystems and algorithms, can be tested by varying the inputs. The inputs can then represent different situations and scenarios that are of interest and which the real-time system should be able to handle.

5.1.2 The RADEx++ Simulator

The RADEx++ simulator models the performance of OR-ULD and OR-ULD/BC in a real-time database system. RADEx++ is an extended ver-

sion of the RADEx simulator [SP94] developed at the University of Massachusetts, Amherst. For a more detailed description of how RADEx++ relates to RADEx, see section C and [Han]. RADEx++ is a stochastic (random input components) and dynamic (evolves over time) discrete-event-based simulator, simulating a centralized active real-time database system. It is primarily designed and used for performance analysis of scheduling, concurrency, and logging algorithms. We will now describe the components of the RADEx++ architecture as shown in figure 5.1.

Workload Control File: This is the configuration file for the simulation, which contains transaction class definitions and a specification of system resources.

Workload Generator: This module simulates the user application of the database system by generating database transactions as specified in the workload control file. Transactions are sent to the database level.

RADEx++ allows the generated workload to consist of multiple transaction classes in contrast to RADEx, where a standard transaction class is described using the following attributes (see appendix C for a more detailed description of transaction class variables):

1. Temporal attributes: arrival rate, periodicity (min., max.), transaction size w_i (min., mean, max.), slack factor, deadline criticality, and deadline tolerance (min., max.).
2. Value functions: type of value function(s), positive utility (min., max.), and negative penalty (min., max.).
3. Other: type of contingency transaction (if any).

A contingency transaction class is specified accordingly:

1. Temporal attributes: transaction size $\bar{w}_i$ (min., mean, max.), and deadline $\bar{d}_i$.[1]

[1] Criticality is inherited by the original transaction. RADEx++ does not support contingency transactions having any deadline tolerance.

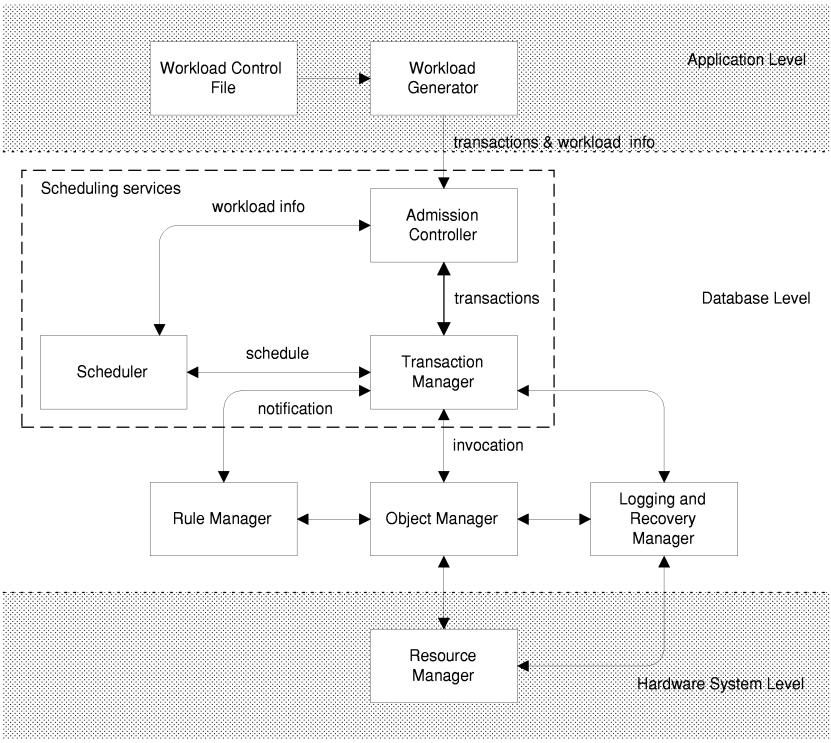


Figure 5.1: RADEx++ simulator components and their interactions

2. Value functions: positive utility (min., max.) (or utility reduction with respect the original transaction), and negative utility (min., max.).

Admission Controller: The admission controller performs overload filtering of transactions arriving to the database level. Admitted transactions are forwarded to the transaction manager. The admission controller exchanges workload information with the scheduler.

Transaction Manager: The transaction manager receives transactions from the admission controller. The transaction manager is respon-

sible for assigning priorities, as determined by the scheduler, to admitted transactions. Moreover, the transaction manager is responsible for the execution of transactions. The transaction manager requests the object manager to execute the methods of the objects accessed by the transactions. Moreover, the transaction manager handles transaction events (BeginOfTransaction, EndOfTransaction, Abort, Commit) and reports them to the rule manager.

Scheduler: The scheduler is responsible for outlining the transaction execution sequence and performing overload resolution if necessary (i.e., in the simulation model the overload resolver is incorporated in the scheduler module). As mentioned earlier, the scheduler interacts with both the transaction manager and the admission controller, exchanging workload information.

Object Manager: The object manager is responsible for controlling concurrency and executing object methods as requested by the transaction manager. Messages are sent to the transaction manager, the rule manager and the recovery manager reporting the status of the execution (transaction method execution successful, transaction method aborted, and transaction method blocked). Additional object events are reported to the rule manager.

Rule Manager: The rule manager receives events from the transaction manager and the object manager. Upon event notification, the rule base is checked to see whether any rules are triggered, and if so condition evaluation is performed. Triggered transactions are generated if the condition is satisfied. The transactions are then submitted to the transaction manager.

Logging and Recovery Manager: The recovery manager is responsible for performing recovery and rolling back of transactions.

Resource Manager: The resource manager simulates the hardware platform, i.e., number of CPUs and their performance, number of disks and their performance etc.

5.1.3 Database System Parameters

We have evaluated the performance of the OR-ULD and OR-ULD/BC algorithms by modeling certain database system parameters, using values for simulation parameters that represent what we believe is realistic and that have been adopted elsewhere (e.g., [HLC91, PLC92, DMK+96, BN96]) as shown in table 5.1. As noted in section 3.2.6, we use dual processors, where one processor is dedicated for scheduling services and the other is dedicated for transaction processing, so the scheduling activities themselves do not influence the ability to meet deadlines.

NumCPU	$1+1$	Number of processing elements
NumDisk	0	Number of disks
ProcOp	10.0	Processing time per database operation (ms)
DBSize	1000	Database size (number of pages)
ArrivalRate	$1-55$	Arrival rate (transactions/sec)

Table 5.1: Database system parameters

5.2 Structure of Simulation Experiments

The behavior of OR-ULD and OR-ULD/BC has been evaluated by running a series of simulation experiments. The purpose of the simulation experiments is: (i) determining the overload performance of the algorithms, and (ii) verifying that the algorithms have predictable behavior. Predictability is expressed in terms of ensuring the meeting of time constraints of critical transactions during transient overloads under a variety of workload conditions.

In order to determine the overload performance of the algorithms, we measure the completion ratios of the transaction classes for a workload

where only the transaction arrival rates are varied. The simulation results are then compared to the results of the well-known EDF algorithm and a baseline algorithm. These experiments are described in section 5.3.1 (OR-ULD) and section 5.4 (OR-ULD/BC).

In order to verify that the behavior of the OR-ULD is predictable for different types of workloads, a series of simulations have been conducted for other transaction workloads. The following workload parameters (in addition to the arrival rate of section 5.1.3) have been varied to simulate different workload scenarios:

- utility of contingency transaction relative to utility of original transaction (section 5.3.2);
- slack factor ($f_i = (d_i - r_i)/w_i$) (section 5.3.3);
- size of transactions (section 5.3.4); and
- size of contingency transactions (section 5.3.5).

5.3 Performance Experiments of OR-ULD

Each experiment was conducted by running a series of three simulations and summarizing them. The primary performance metric is completion ratio (CR), i.e., the ratio of the number of transactions that successfully complete to the total number of transactions requesting resources. Each data point is the average completion ratio with 95% confidence intervals.

The workload consists of two transaction classes (critical class with hard-critical transactions, and non-critical class with firm transactions), where each class represents 50% of the total workload. The critical class consists of sporadic transactions with contingency transactions that differ in size and penalty from the original transaction. Beyond that, the two transaction classes are equivalent. Each transaction class is defined as described in table 5.2 ('←' means that the corresponding value is the same as for the original transaction τ_i):

Transaction Class	Critical		Non-critical
	original	contingency	original
% of workload	50.0%		50.0%
Size (no. of op)	$11 - 15$	$4 - 6$	$11 - 15$
Slack factor	$9.0 - 11.0$	(see footnote 2)	$9.0 - 11.0$
Periodicity	S	$\leftarrow$	A
Utility	$100.0 - 300.0$	$\leftarrow *0.5$	$100.0 - 300.0$
Penalty	$-\infty$	$\leftarrow$	0.0
Write probability	0.25	$\leftarrow$	0.25

Table 5.2: Transaction workload parameters

Aperiodic transactions arrive according to a Poisson distribution. The actual size of a transaction, i.e., the total number of operations it performs, is uniformly distributed within the range as specified by *Size*. Each transaction accesses a number of pages that are selected uniformly within the main-memory-resident database.

5.3.1 Completion Ratio Performance

Our simulations include the results from two related algorithms, namely, pure Earliest Deadline First (EDF) without admission control, and a modified EDF scheduler acting as a comparison baseline (BL) for our experiments. The BL algorithm is based on EDF with an admission controller, where the admission controller changes policy depending on the severity of the overload. Once a transient overload occurs, non-critical transactions are unconditionally rejected by the admission controller, leaving only critical transactions to be admitted. If the workload increases to such extent that critical transactions cannot be admitted based on their original resource requirements, only contingency transactions of critical transactions are admitted from then on. Hence, this algorithm performs two uni-directional mode switches: i) reject all non-critical transactions; and ii) admit only contingency transactions. It

[2]The slack factor is increased in proportion to the change in size between the original transaction τ_i and the contingency transaction $\bar{\tau}_i$.

should be noted that the purpose of BL is to serve as a comparison baseline for our experiments, that is, indicating when a feasible schedule can no longer be found although only contingency transactions are admitted and scheduled. In fact, BL is inappropriate as a scheduling algorithm since it performs the first mode switch when it has failed to admit a transaction independent of its importance and criticality (critical transaction deadlines could be missed). The second mode switch is triggered when an original and critical transaction cannot be admitted. The deadline will be missed for the critical transaction (and possibly a few more).

We have studied the completion ratio of the three algorithms (overload resolution algorithm - denoted OR in the figures, EDF, and BL) as shown in figures 5.2 and 5.3. Figures 5.2(a) and 5.2(b) show completion ratio (CR) as function of total transaction arrival rate for critical and non-critical transactions. Figure 5.3 shows the percentage of critical transactions that have been replaced by contingency transactions (replacement ratio RR).

EDF starts missing transaction deadlines when the transaction arrival rate exceeds approximately five transactions per second, which is expected since this represents a workload utilizing approximately eighty percent. The inability of EDF to handle overloads, causing a domino effect of missed deadlines, is well known (also shown in figure 5.2). Since both classes share the same characteristics from the viewpoint of EDF, no transaction class is favored. Thus, the decay rates of the completion ratio are similar.

It is of paramount importance that time constraints of critical transactions are met, and it can be observed that OR-ULD satisfactorily enforces this requirement within a wide operational envelope (in this case up to 33 transactions/second).[3] OR-ULD performs close to optimal during extreme overloads when compared to the baseline BL in terms of meeting time constraints for critical transactions. OR-ULD ensures the timeliness of critical transactions by gradually increasing (from zero) the numbers

[3]By operational envelope we mean workload conditions under which correct behavior is maintained, i.e., critical time constraints and $MCCR_i$ constraints are enforced.

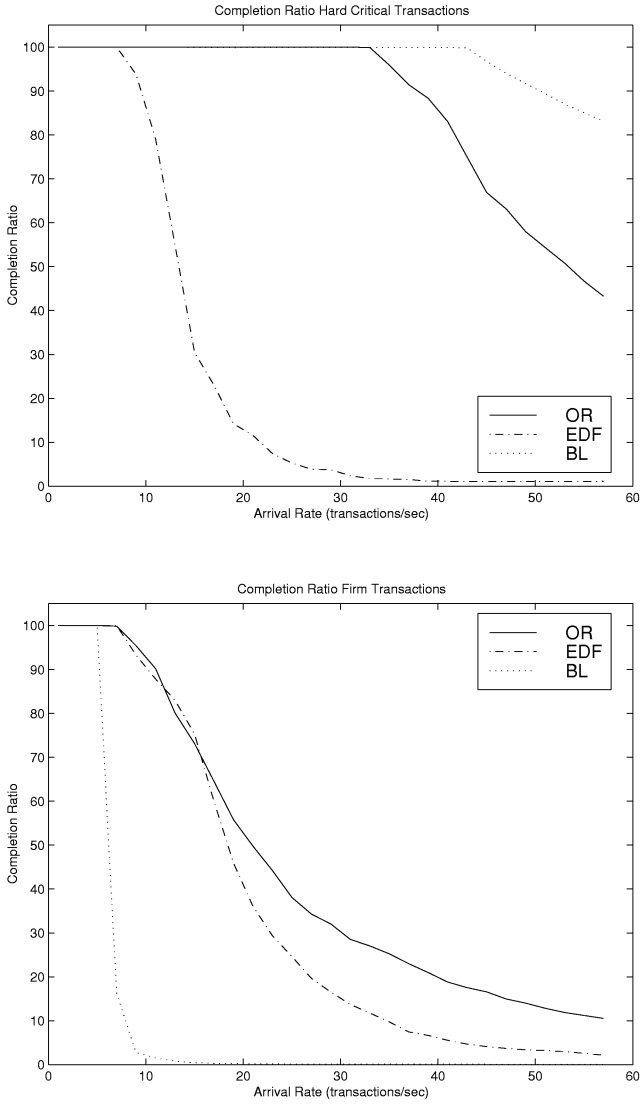


Figure 5.2: Performance analysis of OR-ULD compared to EDF and the baseline, showing CR for (a) critical [top] and (b) non-critical [bottom] transactions

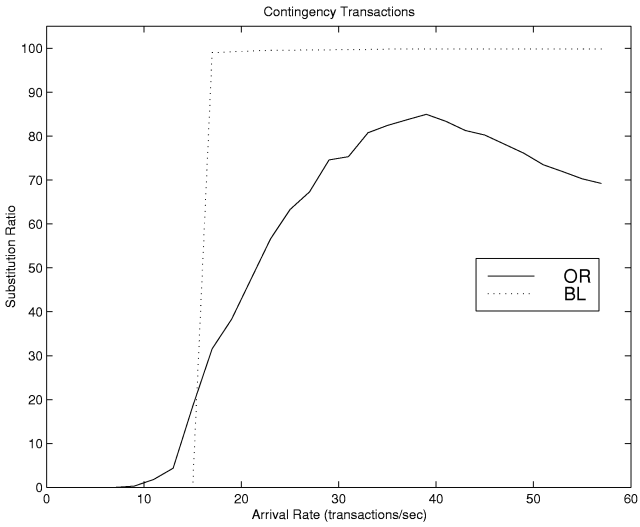


Figure 5.3: Performance analysis of OR-ULD compared to EDF and the baseline, showing RR for contingency transactions

of critical transactions being replaced by their contingency transactions (see figure 5.3), and gracefully dropping/rejecting non-critical transactions. This produces a significant improvement compared to both the EDF (in terms of guaranteeing critical transactions at the expense of non-critical ones) and the baseline (in terms of executing a large number of non-critical transactions also under mild overload conditions) algorithm.

The reason why OR-ULD does not achieve optimal results during extreme overloads, as set out by the by baseline, is the sporadic nature of the workload and the fact that the dynamic scheduler is not clairvoyant. Original transactions are admitted when possible since they contribute a high utility to the system. The drawback of this is that when additional transactions arrive, a feasible schedule may no longer be found. Although original transactions are replaced by their contingency transactions, unnecessary processing time has been lost. This can be overcome to some

extent by having the workload monitored, and when the workload goes beyond a certain level the admission controller is notified to only admit contingency transactions. (Current work focuses on predicting transient overloads in dynamic real-time systems using artificial neural networks [MS99, HSZ99]).

5.3.2 Varying the Utility of Contingency Transactions ($\bar{v}_i(t)$)

In this experiment, the benefit contributed by contingency transactions was varied. Other workload parameters remained unchanged as specified in table 5.2.

The utility $\bar{v}_i(t)$ contributed by contingency transactions was varied by letting $\bar{v}_i(t)$ be a function of $v_i(t)$. Specifically, $\bar{v}_i(t)$ equals $c * v_i(t)$, where c was varied between 0.0 (no benefit is obtained for completing contingency transactions $\bar{\tau}_i$) and 1.0 (there is no reduction in benefit when completing contingency transactions as opposed to original transactions) by increments of 0.1.

The results, shown in figures 5.4 and 5.5, show that the completion ratio of critical transactions is intact (for arrival rates within the operational envelope). As figure 5.5(a) shows, the completion ratio of original transactions is low for high values of c, which is expected. As the difference in utility contributed by contingency transactions and original transactions decreases, the utility loss of replacing original transactions decreases as well. Hence, at high values of c, OR-ULD more easily favors early replacement of critical transactions as opposed to dropping non-critical transactions. This is confirmed by figure 5.5(b). As a natural consequence, the completion ratio of non-critical transactions increases as c increases (see figure 5.4(b)).

In this experiment, there is a significant increase in original transactions being replaced for c values higher than 0.6 as seen by the steep increase in figure 5.5(b). OR-ULD is utility loss density driven and, hence, the breaking point where the replacement ratio starts increasing significantly

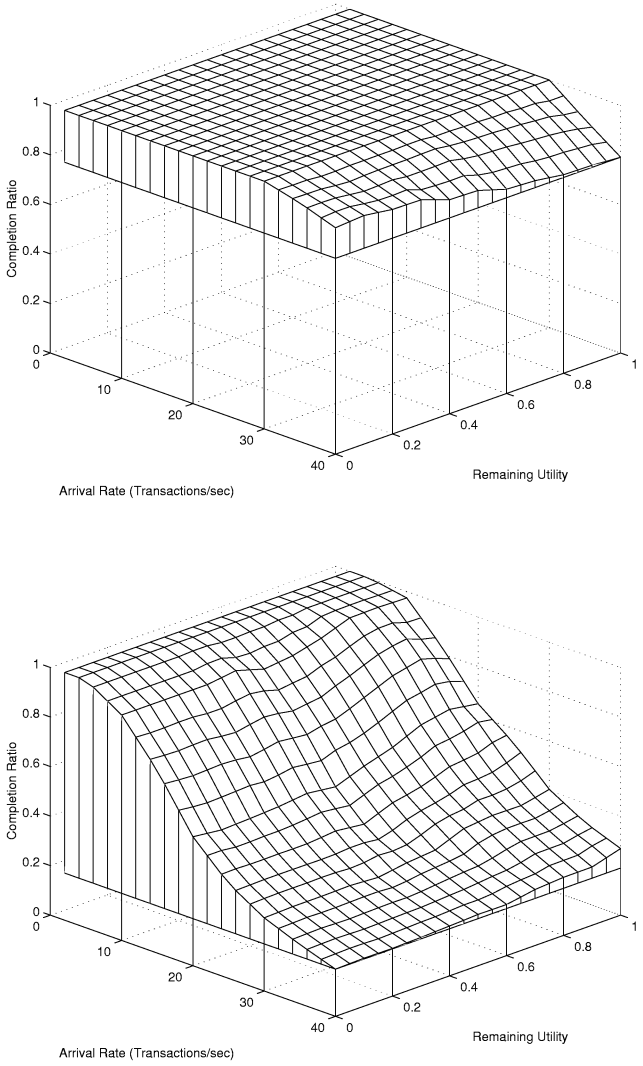


Figure 5.4: Simulation results showing CR for (a) critical [top] and (b) non-critical [bottom] transactions when varying $\bar{v}_i(t)$

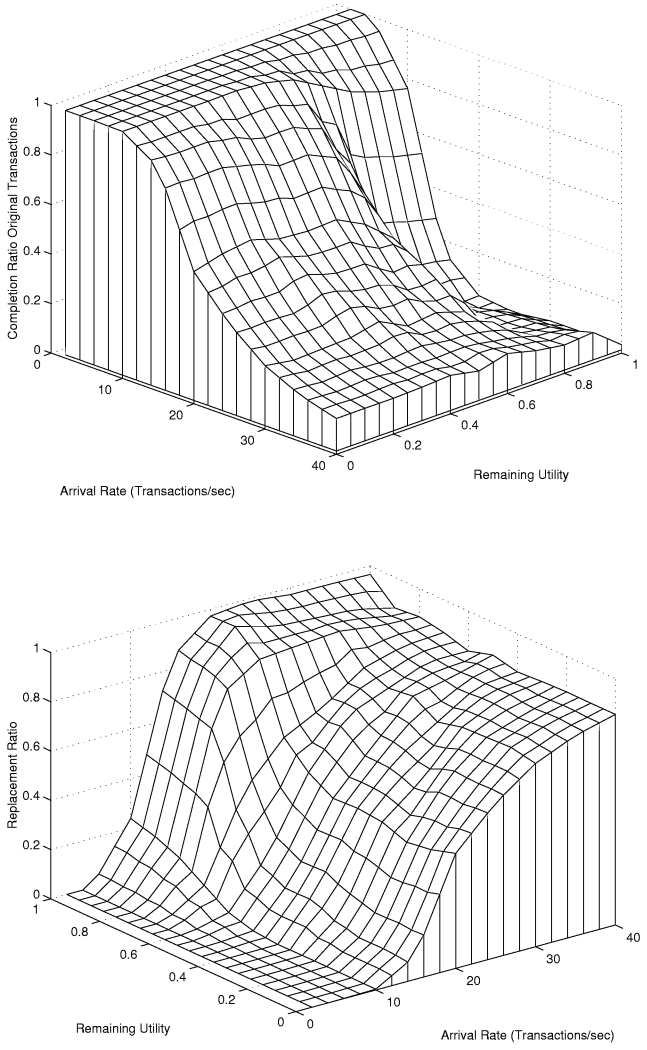


Figure 5.5: Simulation results showing (a) CR for original transactions [top] and (b) RR for contingency transactions [bottom] when varying $\bar{v}_i(t)$

may vary since it depends on the size of contingency transactions as well as the utility contributed by non-critical transactions.

5.3.3 Varying the Slack Factor

In this experiment, the slack factor of the transaction was varied. The slack factor, denoted f_i, is defined as the ratio between the difference of the deadline and the ready time, and the worst-case execution time of transaction τ_i, i.e., $f_i = (d_i - r_i)/w_i$. The average slack factor f for transactions was varied between 2.0 and 15.0 by increments of 1.0, and was uniformly distributed within the range $f * [0.75, 1.25]$. (Since w_i are fixed, d_i are assigned accordingly). The other workload parameters remained unchanged as specified in table 5.2. Note that the average slack factor starts at 2.0 in order to give the scheduler sufficient flexibility. (For example, a transaction with slack factor 1.0 would imply that the transaction must start once it arrives and its execution must not be interrupted in order to meet its deadline, i.e., there is no flexibility in the schedule).

Here figures 5.6 and 5.7 show that the OR-ULD behavior is, with the exception of low values of f, independent of the slack factor. The figures show that there is virtually no change in completion ratio and replacement ratio for slack factor values greater than 3.0. There is a small decrease in percentage of completed non-critical transactions when the slack factor is less than 3.0, i.e., tight time constraints. In this case non-critical transactions are dropped or rejected at very low workloads (see figure 5.6(b)). A small decrease in completion ratio of critical transactions can also be observed, i.e., the operational envelope shrinks (see figure 5.6(a)). The decrease in performance when scheduling transaction workloads with tight time constraints, i.e., low slack factors, is expected. As the slack factor of the transactions decreases, the risk that the execution of a transaction may jeopardize another transaction increases. As a consequence in our case, the search for cost efficient overload resolution plans becomes harder.

In conclusion, the results show that the predictability of OR-ULD is not

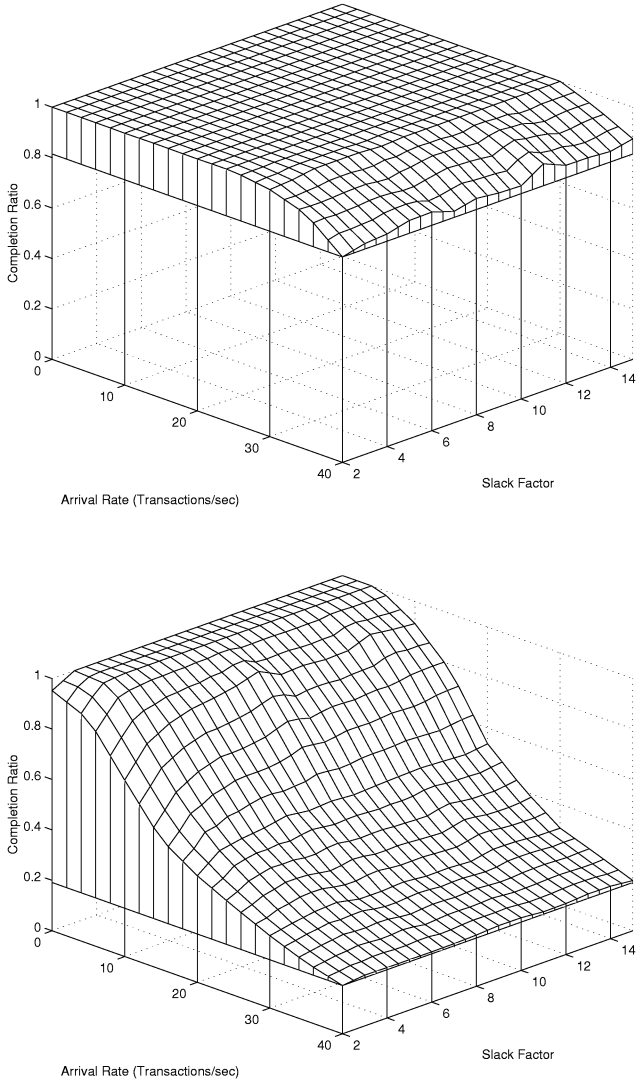


Figure 5.6: Simulation results showing CR for (a) critical [top] and (b) non-critical [bottom] transactions when varying slack

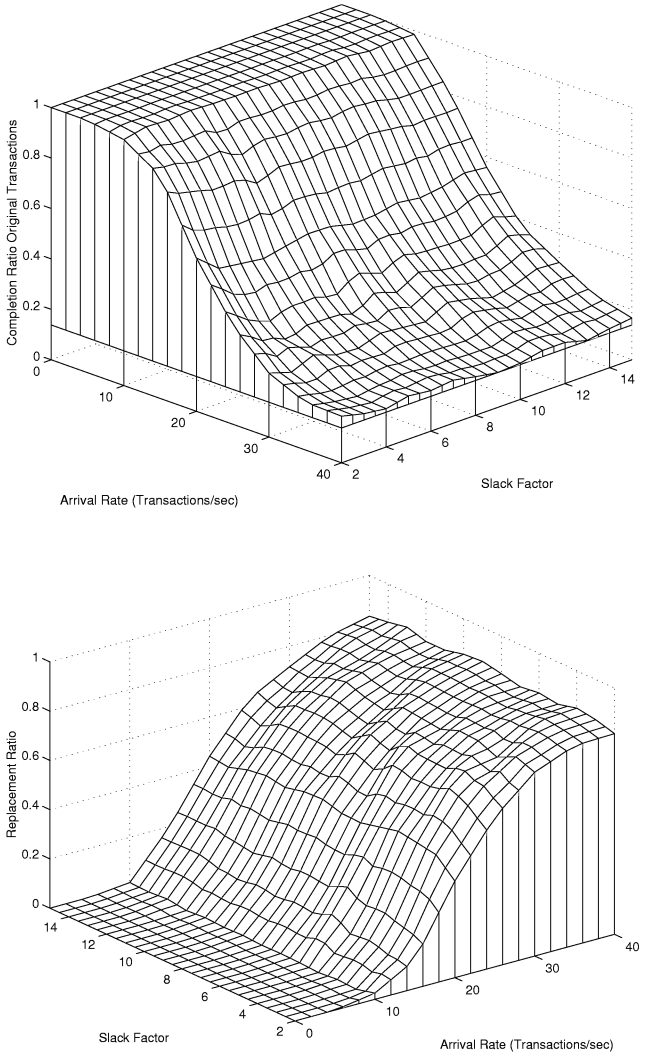


Figure 5.7: Simulation results showing (a) CR for original transactions [top] and (b) RR for contingency transactions [bottom] when varying slack

affected when the slack factor of transactions is varied. However, a small decrease in performance can be noted for small values of f ($f < 3.0$), in which case primarily non-critical transactions are dropped/rejected.

5.3.4 Varying the Size of Transactions (w_i)

In this experiment, the size of transactions (w_i), i.e., the number of operations, was varied, while remaining workload parameters as specified in table 5.2 were kept unchanged. The average number of operations, denoted w_{avg}, of original transactions was varied between 10.0 and 25.0 by increments of 2.0, and individual sizes were uniformly distributed within the range $w_{avg} * [0.75, 1.25]$. The change in size of contingency transactions remained proportional to the change in size of the original transaction, i.e., their size ratio was kept constant throughout the simulations.

Here figures 5.8(a) and 5.8(b) show that the completion ratios for both critical (outside the operational envelope) and non-critical transactions are reversely dependent on the average transaction size. Hence, the completion ratio of the transactions decreases as the average transaction size increases. This behavior is expected and explained by the increased resource requirements demanded by both original transactions and contingency transactions. The rate at which contingency transactions are invoked increases with the arrival rate and the transaction size (see figures 5.9(a) and 5.9(b)). The reason that the overall completion ratio decreases with the arrival rate and the transaction size is primarily due to the increased size of the contingency transactions. (If the size of contingency transactions were the same for all workloads, the overall completion ratio would not decrease).

5.3.5 Varying the Size of Contingency Transactions ($\bar{w}_i$)

In this experiment, the size of contingency transactions was varied while remaining workload parameters were kept unchanged as specified in table 5.2. The average size of the contingency transactions, denoted $\bar{w}_{avg}$,

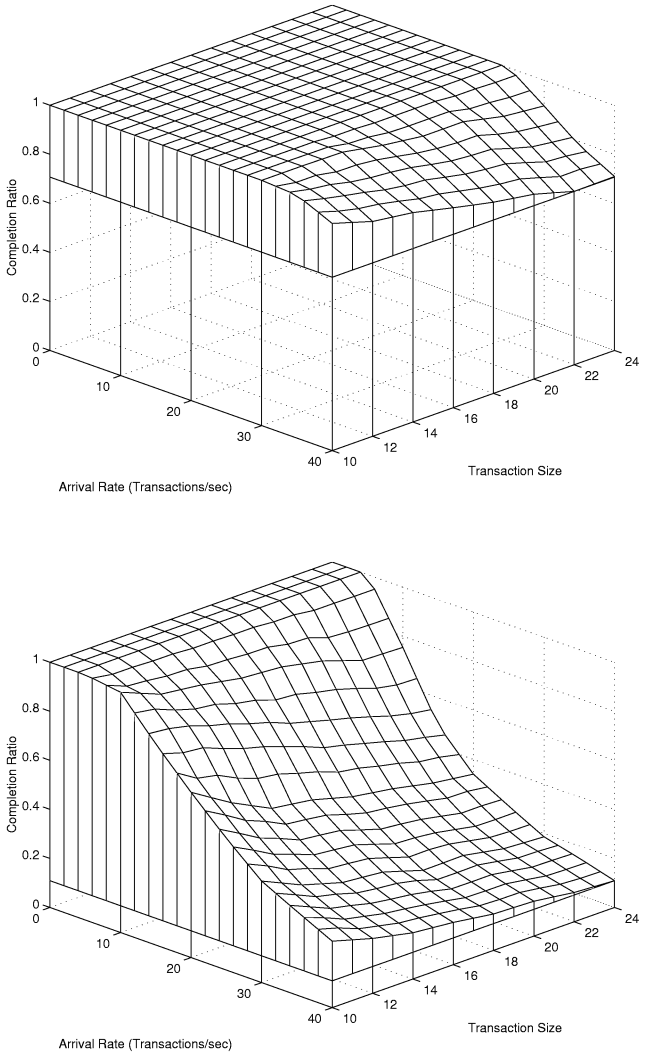


Figure 5.8: Simulation results showing CR for (a) critical [top] and (b) non-critical [bottom] transactions when varying w_i

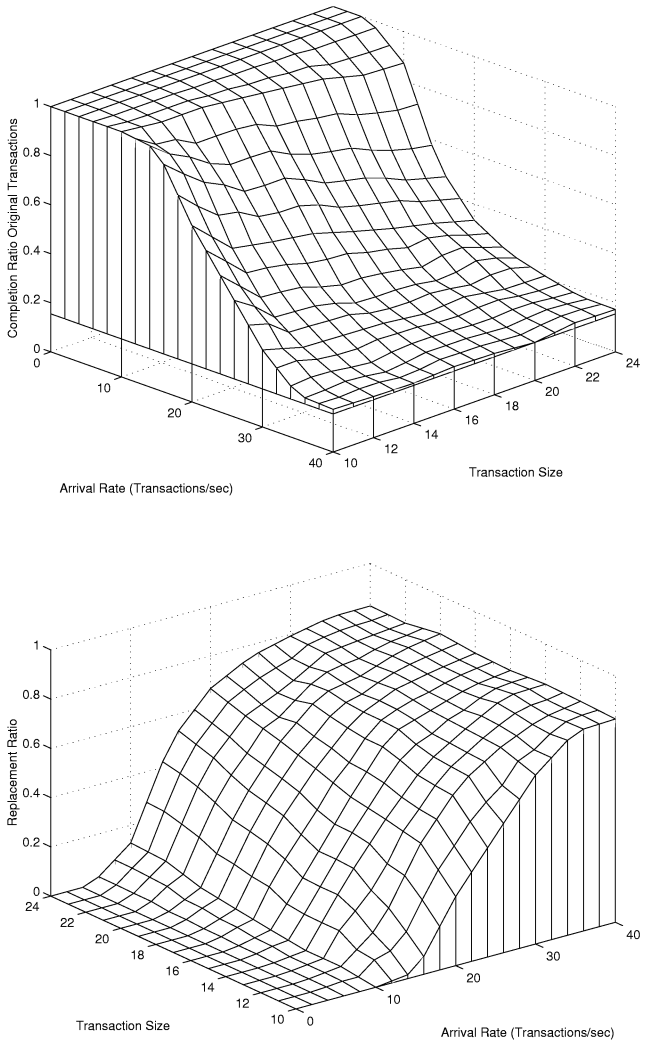


Figure 5.9: Simulation results showing (a) CR for original transactions [top] and (b) RR for contingency transactions [bottom] when varying w_i

was varied from 4.0 to 12.0 by increments of 2.0, and the individual sizes of the contingency transaction were uniformly distributed within the range $\bar{w}_{avg} * [0.75, 1.25]$ (and given that $\bar{w}_i \leq w_i$ is true). The size of original transactions and non-critical transactions remained unchanged ($[11.0, 15.0]$), implying that the utility loss density for dropping a transaction remained unchanged. However, ORAs replacing original transaction with contingency transactions are less likely to be executed as the size of the contingency transaction increases. As the sizes of the contingency transactions increase, the amount of time saved by invoking a contingency transaction decreases and, hence, the utility loss density increases (making the ORA less desirable).

Our simulation studies show that the completion ratio of critical transaction decreases, in a predictable manner, as the size of the contingency transaction increases (see figure 5.10(a)). However, the completion ratio of non-critical transactions is, in this workload scenario, virtually unchanged and stable (see figure 5.10(b)). The results show that the completion ratio of original transactions is to some extent independent of the size of the contingency transactions (see figure 5.11(a)).

For moderate overloads, we notice that as $\bar{w}_i$, the decay rate of the completion ratio of original transactions reverses. Figure 5.11(b) shows that contingency transaction invocation increases as $\bar{w}_i$ increases. This may seem counter-intuitive, since large contingency transactions have high utility loss density, indicating that very limited time will be saved relative to the utility loss and, hence, normally the number of invocations should decrease. However, as $\bar{w}_i$ increases, the number of critical transactions that need to be replaced in order to de-allocate the same amount of resources increases. Hence, the ORPs tend to be more complex and consist of more ORAs. In this particular case, transactions that are more critical are replaced by their contingency transactions already at admission time.

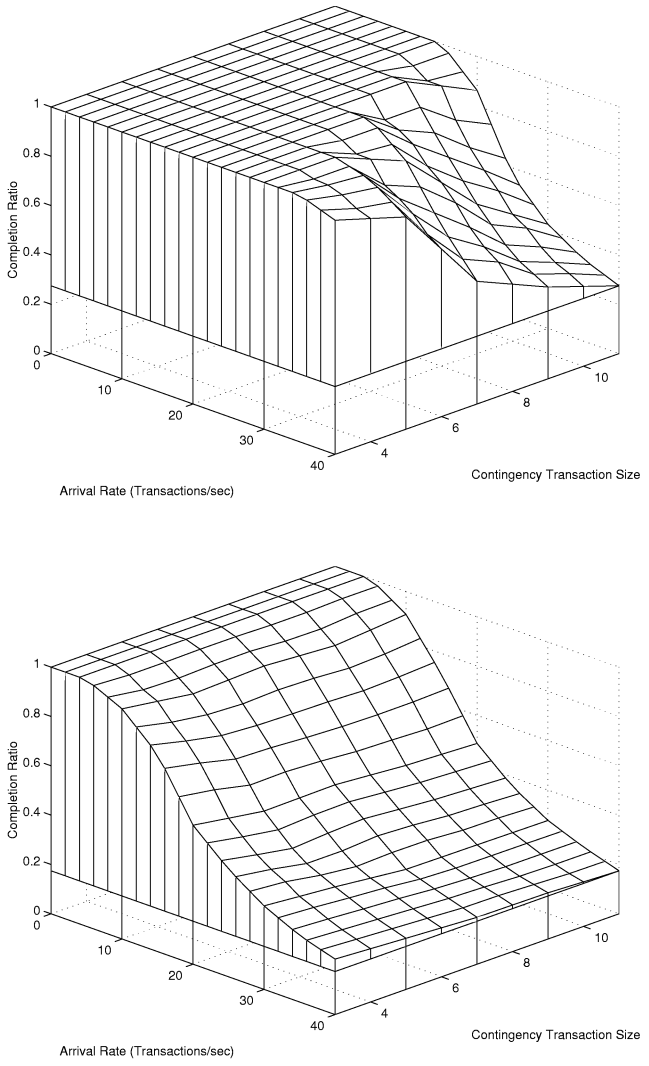


Figure 5.10: Simulation results showing CR for (a) critical [top] and (b) non-critical [bottom] transactions when varying $\bar{w}_i$

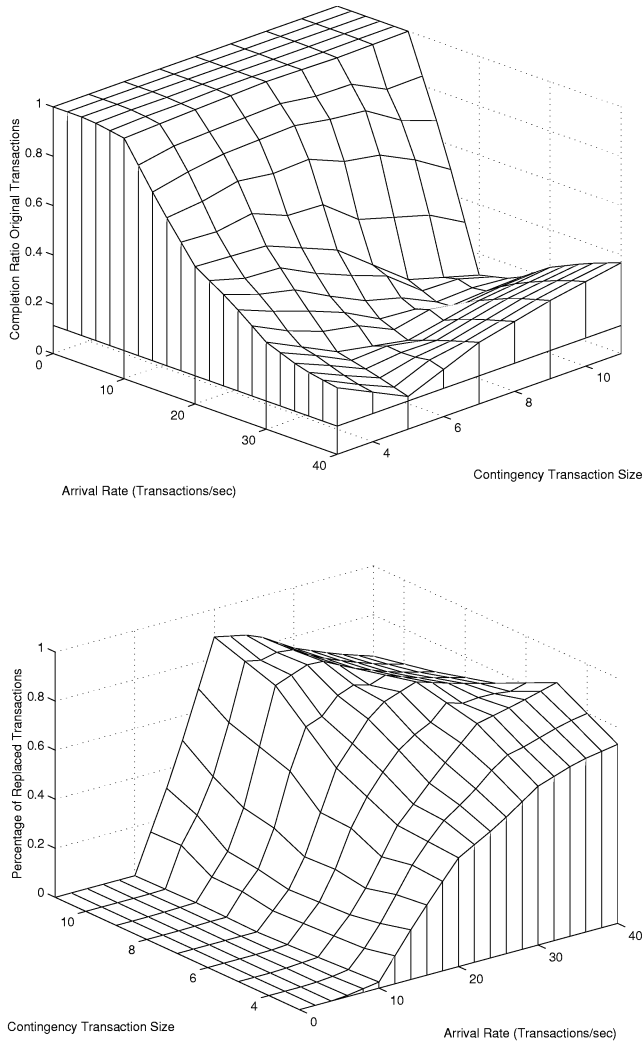


Figure 5.11: Simulation results showing (a) CR for original transactions [top] and (b) RR for contingency transactions [bottom] when varying $\bar{w}_i$

5.4 Performance Experiments of OR-ULD/BC

With OR-ULD/BC we performed an experiment with a non-critical transaction workload only, and an experiment with a complex (mixed) workload.

In the first experiment with bias control (/BC), the workload consisted of three non-critical transaction classes only. In the second such experiment, the workload consisted of one critical transaction class and two non-critical transaction classes. We compare the results of OR-ULD/BC with the results of OR-ULD and EDF.

Each experiment was conducted by running a series of simulations and summarizing them. Again, the primary performance metric is completion ratio (CR), i.e., the ratio of the number of transactions that successfully complete to the total number of transactions requesting resources. Each data point in the experiments represents the average completion ratio with 95% confidence intervals. The EDF scheduler is here used as the baseline.

5.4.1 Non-Critical Transaction Workload

The workload in this experiment was defined to consist of three non-critical transaction classes, where each class represents one third of the total workload. Two transaction classes had minimum class completion ratio (MCCR) constraints (see table 5.3). Each transaction class was defined as follows (the symbol '←' (used in the tables) means that the value equals the value in the adjacent column).

Here figure 5.12(b) shows the results from the EDF algorithm, and as expected, the decay rates of the completion ratios are similar for all transaction classes.[4]

As figure 5.12(a) shows, the bias control mechanism in OR-ULD/BC has the desired effect, that is, of favoring transaction classes having con-

[4]Arrival rates in the graphs denote the total number of transactions arriving to the system per second.

Transaction Class	Class 1	Class 2	Class 3
% of workload	33.3%	33.3 %	33.3 %
Size (no. of ops)	$5 - 9$	←	←
Slack factor	$9.0 - 11.0$	←	←
Utility	$50.0 - 100.0$	←	←
MCCR	0.75	0.25	0.0
Write probability	0.25	←	←

Table 5.3: Transaction workload parameters

straints on the completion ratio. The simulations show that CCR_1 and CCR_2 decrease linearly more or less, while CCR_3 decreases exponentially. OR-ULD/BC enforces robustness requirements and finds feasible schedules (satisfying the MCCRs) until the workload goes beyond 41 transactions/second. However, as the figure shows, although the completion ratio constraints have been violated for the first and second transaction class, a small percentage of the third class transactions are still successfully completing. Ultimately, transactions having no completion ratio constraints should not be given system resources when the system is dealing with extreme overloads. Instead, system resources should be granted to transactions with completion ratio constraints. However, this effect is due to the lack of clairvoyance, and the inherent deficiency in bias control (bias does not become infinite when MCCRs are violated).

5.4.2 Complex Transaction Workload

In this simulation experiment the workload consisted of three transaction classes, where each class represents one third of the total workload. The first transaction class consisted of sporadic critical transactions, where each transaction has a corresponding contingency transaction. The other two classes consisted of aperiodic non-critical transactions, both having completion ratio constraints (see table 5.4 for transaction class definitions). OR-ULD (figure 5.13(b)) and EDF (figure 5.14) are here used as baselines.

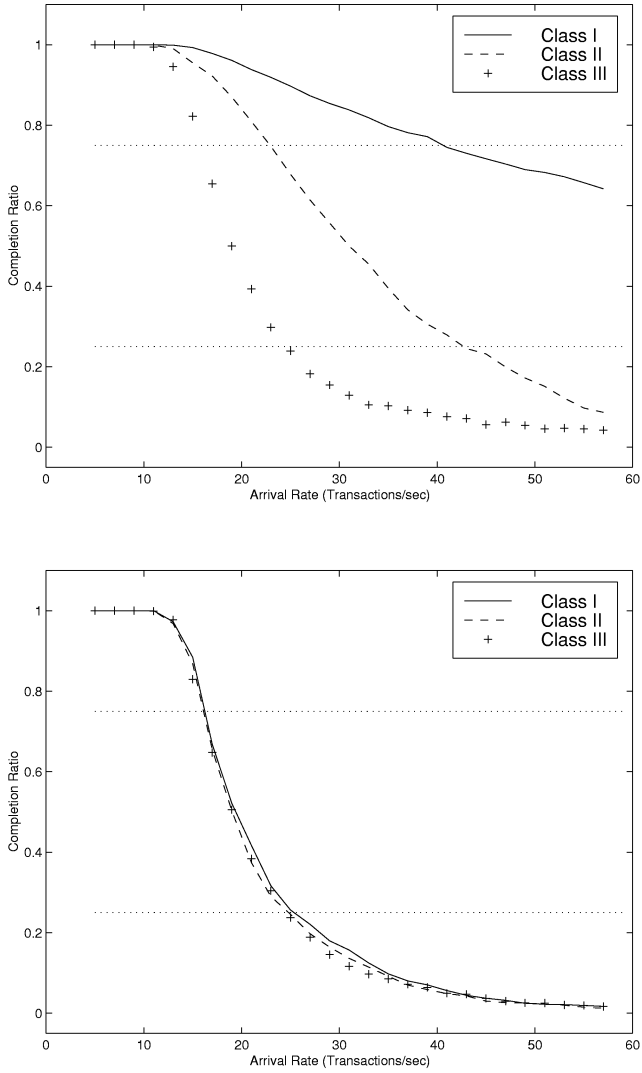


Figure 5.12: Simulation results showing CR for (a) OR-ULD/BC [top] (b) EDF [bottom]

Transaction Class	Class 1		Class 2	Class 3
	original	contingency	original	original
% of workload	33.3%		33.3%	33.3%
Size (no. of ops)	$5-9$	$4-6$	$5-9$	$\leftarrow$
Slack factor	$9.0-11.0$	(see footnote 5)	$9.0-11.0$	$\leftarrow$
Utility	$50.0-100.0$	$\leftarrow *0.5$	$50.0-100.0$	$\leftarrow$
Penalty	$-\infty$	$\leftarrow$	0.0	$\leftarrow$
MCCR	N/A		0.50	0.25
Write probability	0.25	$\leftarrow$	$\leftarrow$	$\leftarrow$

Table 5.4: Transaction workload parameters

Here figure 5.13(a) shows that OR-ULD/BC enforces the time constraints of critical transactions, which is expected given earlier results using OR-ULD. Transient overloads are resolved by increasing the number of contingency transactions that replace critical transactions (class 1) as the load grows (see the 'o'-line in figure 5.13(a)), and dropping non-critical transactions (class 2 and 3) gracefully. For this specific transaction workload, the resolver starts dropping non-critical transactions and replacing critical transactions with contingency transactions at about the same time. However, as shown in figure 5.13(a), the rate at which transactions are replaced increases significantly at low levels. This behavior is expected given the low utility loss imposed by replacing a critical transaction in contrast to dropping a non-critical transaction. Out of the original transactions, at most 93% are replaced during the transient overload, with the peak at 29 transactions/second. The completion ratio of the second transaction class drops below its minimum level for arrival rates higher than 33 transactions/second. The completion ratio of the third transaction class is then about five percentage units above the minimum requirement for the transaction class. However, while OR-ULD violates the completion ratio constraints for transaction workloads exceeding 27 transactions/second, OR-ULD/BC ensures completion ratio constraints for workloads less than 34 transactions/second. Hence, it shows that the

[5]The slack factor is increased in proportion to the change in size between the original transaction τ_i and the contingency transaction $\bar{\tau}_i$.

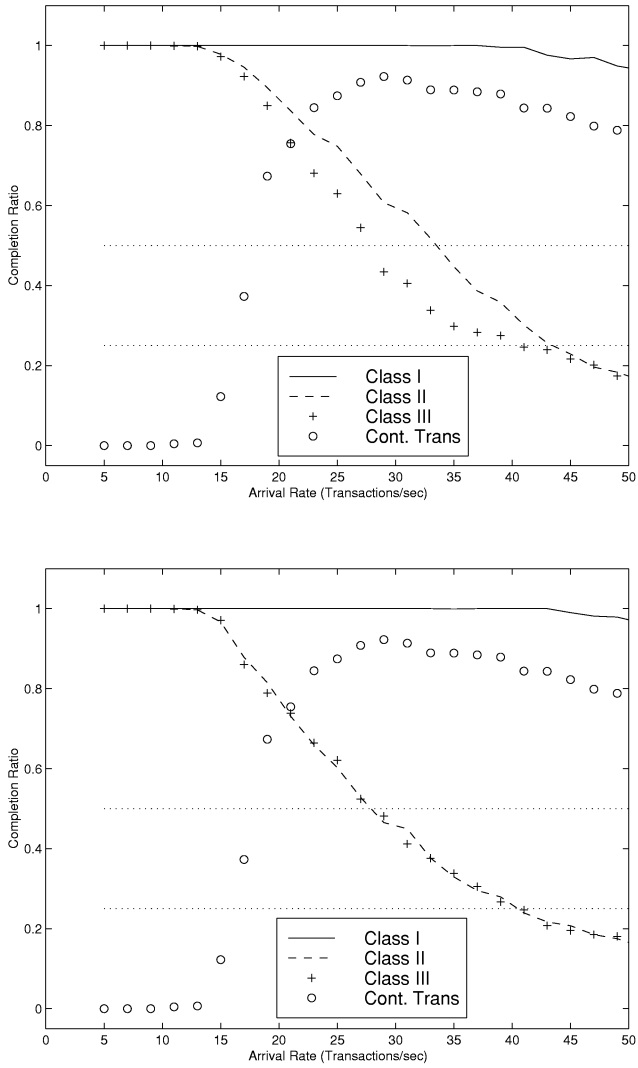


Figure 5.13: Simulation results showing CR for (a) OR-ULD/BC [top] and (b) OR-ULD [bottom]

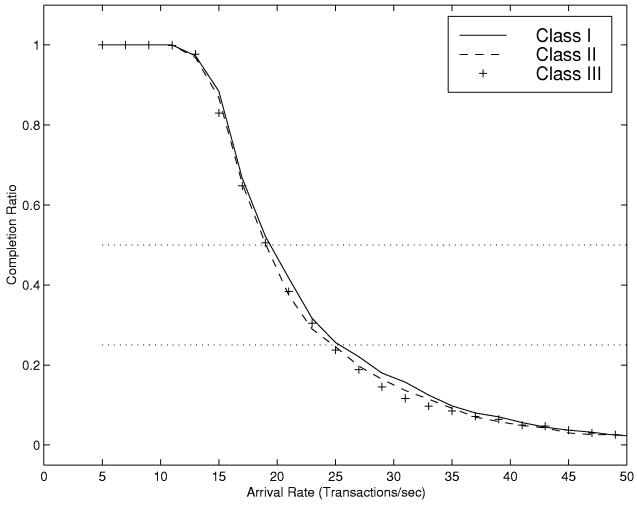


Figure 5.14: Simulation results showing CR for EDF

bias control mechanism is able to favor transaction classes with completion ratio constraints. However, critical transactions start missing their deadlines 39 transactions/second. Overall, the envelope is 33 transactions/second.

5.5 Guidelines for Assigning Value Functions

In any utility-driven approach the assignment of value functions to transactions is imperative. In our simulation studies, we have intentionally used a workload where the positive utility contributed by critical transactions and non-critical transactions have been the same, thus complicating the overload resolution. As our simulations show, OR-ULD guarantees the time constraints of critical transactions by, in most cases, gradually replacing transactions having contingency transactions and gracefully dropping non-critical transactions.

Based on our experience with OR-ULD, the following guidelines are rec-

ommended in order to ensure graceful performance degradation during transient overloads, and to enforce the timeliness of critical transactions.

- Since OR-ULD is driven by utility loss, it is required that transaction criticality is modeled and represented with negative utility (penalty) in the value functions (as opposed to other, more artificial representation techniques). This ensures that overload resolution actions that suggest dropping critical transactions will not be selected.

- Transactions must have some degree of slack, allowing OR-ULD some flexibility in resolving transient overloads and outlining a schedule. (In our simulation studies, OR-ULD had predictable behavior for slack values higher than 3.0.)

- A small difference in utility contributed by a transaction and a contingency transaction ($v_i(t) \sim \bar{v}_i(t)$), in comparison to the utility contributed by a non-critical transaction, favors early replacement of original transactions. This is particularly true if there is a significant difference in worst-case execution times ($w_i(t) \gg \bar{w}_i(t)$). If it is desirable to have early dropping of non-critical transactions under mild overload conditions, and late replacement of original transactions (heavy overload), then the difference in benefit of original transactions and their contingency transaction should significantly exceed the benefit of non-critical transactions.

Chapter 6

Related Work

Real-time scheduling problems have been studied for several decades. It is beyond the scope of this work to relate our research to all real-time scheduling research efforts that have been made. This chapter focuses on research problems and algorithms that are most relevant to the work in this thesis, primarily dynamic real-time scheduling algorithms. For more comprehensive studies and surveys on real-time scheduling in general, the reader is recommended to study the work by Burns et al. [Bur91], Stankovic et al. [SSDNB95, SSRB98], Buttazzo [But97], and Parnas and Xu [XP93].

This chapter is structured as follows. In section 6.1 and section 6.2 other work done on value-driven scheduling and deadline-driven scheduling is presented. In section 6.3 we elaborate on some static scheduling models. Section 6.4 discusses work done on bias control. In section 6.5, we discuss the similarities and differences between OR-ULD and other approaches based on alternative action models. Section 6.6 concludes the chapter with a tabular overview comparing OR-ULD to other approaches.

6.1 Dynamic Value-Driven Scheduling

Initial work on value-driven scheduling was carried out by Locke [Loc86], who performed a thorough analysis of best-effort scheduling of tasks having value functions. The problem was to maximize the total utility given to the system when scheduling independent and non-precedence-constrained preemptable tasks having firm or soft deadlines, where preemption is of the type preemption-restart. Locke's view is that the notion of a single deadline is not adequate in a value-driven approach. Instead, discontinuities in the value functions, or their derivatives, are considered critical time-points. His study excludes the case where tasks impose a negative value on the system after the ready time of the tasks and, hence, these tasks are considered non-critical.

Locke's computer system model assumes a shared-memory multiprocessor environment, where tasks are memory resident. The scheduler is assumed to execute on a separate processor on which no tasks are to be executed. In his model, value functions can have one of the following characteristics: linearly decreasing, linearly increasing or constant between the ready time and the deadline. After the deadline, the value decays exponentially. The Best-Effort scheduling algorithm, developed by Locke [Loc86], showed through extensive simulation and analysis the feasibility of using dynamic scheduling algorithms for soft real-time scheduling problems.

Tseng et al. [TCY97] conducted a simulation and performance analysis for scheduling value-based firm deadline transactions in fully and partially main-memory-resident database systems. The value contributed to the system by the individual transactions is represented as a step function. A step function is piecewise constant (all functions return a constant value). That is, the time of completion does not affect the total utility, unless a deadline is missed in which case no value (benefit) is given to the system. The study included the following algorithms: Earliest Deadline First (EDF), Highest Value First (HVF), Value-Inflated Deadline (VID), Value-Inflated Relative Deadline (VIRD) [HCL93], and Highest Reward First (HRF) [TCY97]. The concurrency control protocol

used was 2PL-HP (Two-Phase Locking with Highest Priority) [AGM88].

Chen and Muhlethaler [CM96] developed two value-driven scheduling algorithms for non-preemptive tasks with non-critical deadlines, with the objective to maximize total value. To achieve this, the algorithms adopt both heuristics and decomposition. Their simulation studies show that the suggested algorithms outperform the benchmark algorithm EDF[1] [CM96], which is not surprising since the EDF algorithm is not value-cognizant.

When comparing our proposed approach with OR-ULD to these other dynamic value-driven schemes [Loc86, HCL93, TCY97, CM96], some significant differences can be noted. Most importantly, OR-ULD is designed to schedule transactions in real-time database systems where transactions are assumed to be of different criticality and importance. Moreover, critical transactions have contingency transactions that can be invoked during transient overloads

6.2 Dynamic Deadline-Driven Scheduling

The EDF algorithm has been shown to be optimal when scheduling preemptable tasks with arbitrary release times and deadlines [LL73, Der74, MD78], i.e., if a feasible schedule exists, then EDF will find it. It is well known that EDF exhibits poor behavior during overloads. We now study some of the research that has been carried out on developing other deadline-driven algorithms and extending the EDF algorithm to handle different types of scheduling problems.

Abbott and Garcia-Molina [AGM88] evaluated the EDF and Least Slack scheduling algorithms for firm-deadline transactions in a main-memory-resident database system. The scheduling algorithms are evaluated in combination with three different policies for determining the eligibility of transactions. In addition, three different concurrency control schemes were tested, namely, serial execution (non-concurrent), high priority, and

[1] Two sets of EDF algorithms were implemented, namely ESDF (soft deadlines) and EHDF (hard deadlines).

conditional restart.

Buttazzo and Stankovic [BSS95] investigated the overload performance of the EDF, HVF (see section 6.1), HDF (Highest Density First; priority is based on value and execution time (v_i/w_i)), and the *Mix* dynamic scheduling algorithms. The Mix algorithm considers both importance value and deadline when determining the task priority. Two derivatives of each algorithm were developed to enforce the notion of guarantee and robustness. The guarantee algorithms use an acceptance test invoked at the time of activation of new tasks. The robustness algorithms, in addition to performing a guarantee test upon task activation, have a more sophisticated rejection strategy. Tasks have a value reflecting the importance level of the individual task, and their work suggests that scheduling by deadline and rejecting by value is the most effective strategy for a wide range of overload conditions. A rejection strategy based on the importance value of the tasks is adopted and combined with a resource reclaiming mechanism taking advantage of early task terminations [BSS95].

Buttazzo and Stankovic developed the RED (Robust Earliest Deadline) scheduling algorithm [BS93, BS95] which tolerates overloads by using an acceptance test. If the new task set is not feasibly scheduled, then the system will reject the task with the least value, if any, such that the remaining set is schedulable (otherwise the new task is rejected). All tasks are aperiodic with firm and soft deadlines with specified deadline tolerance.[2] Similar to [BS93, BSS95, BS95], our proposed overload resolution strategy admits transactions based on deadline, and rejection is based on value.

Spuri et al. [SBS95] implemented a server, called Total Bandwidth Server (TBS), which served aperiodic tasks with soft and firm deadlines (periodic requests were considered critical). However, firm and soft tasks may be rejected at admission during overloads.

[2]The terminology used in this thesis defines a deadline that has some tolerance to be soft. A non-critical transaction with no deadline tolerance is firm. Using this terminology, RED handles both soft and firm deadline transactions, although the authors refer to the deadlines as firm in the articles.

Baruah et al. [BKM+91] developed a scheduling algorithm called D* (based on EDF) for scheduling firm-deadline tasks on a single processor. D* is optimal, in the sense that it gives the best competitive factor possible, during non-overloads like EDF and LS.

Haritsa et al. [HLC91] developed two new EDF-based algorithms for scheduling firm-deadline transactions in real-time database systems having multiple processors. The AED (Adaptive Earliest Deadline) algorithm adapts its behavior based on the workload by using a feed-back control process. On the basis of previous experience the process estimates the number of transactions that are sustainable under EDF. While AED is a strict deadline-driven algorithm, the HED (Hierarchical Earliest Deadline) algorithm is value-driven and attempts to minimize the value loss, i.e., maximizing the hit ratio (hit ratio is the ratio of the actual and estimated number of transactions that successfully completed). In both cases, transactions are assumed to have firm deadlines, and their worst-case execution times are not known a priori. In addition, transactions have importance values (used only with HED).

The approaches described in [BS93, BS95, BSS95, SBS95, BKM+91] share similar characteristics to the research problem studied in this thesis. For example, worst-case execution times of transactions are known a priori, while arrival times and deadlines are not. Moreover, transactions have value functions reflecting importance that do not change over time, and the algorithms are designed for maximizing the total value (hit value ratio, i.e., the ratio of obtained value and the obtainable value), by rejecting less important tasks during overloads. In contrast to all of these approaches OR-ULD copes with additional complexity with respect to the multi-class nature of the workload (critical and non-critical transactions), and the additional transaction element, namely the contingency transaction. This requires more sophisticated admission control and rejection algorithms. Our strategy enables overload resolution by rejecting transactions at admission time, dropping selected admitted transactions, or replacing transactions with their contingency transactions.

Sivasankaran et al. [SST+96] developed and evaluated, by extensive simulation, three deadline-driven algorithms suitable for active real-time

main-memory-resident database systems. The system has two transaction classes distinguished by their reactive behavior: non-triggering transactions and triggering transactions having immediate or deferred coupling to the "triggering transaction".[3] Transactions have known worst-case execution times and firm deadlines. They show that the completion ratio for triggering transactions is increased significantly when priority assignment policies take into account the dynamically generated workload. Moreover, dynamically evaluating and changing the priorities of the transactions depending on their behavior with respect to triggering rules, results in a significantly increased completion ratio. In comparison, our proposed strategy with OR-ULD does not consider the multi-processor case. Our proposed overload resolution strategy incorporates value functions and value driven overload management. Moreover, critical transactions have contingency transactions that are exclusively coupled to the original transactions. OR-ULD also allows for different types of time constraints.

6.3 Static and Hybrid Scheduling

In this section we compare OR-ULD to static and hybrid approaches. In our comparison we consider the nature of the workload (single-/multi-class workload, arrival patterns) and tasks (critical/non-critical).

Liu et al. [LL73] developed the rate-monotonic scheduling algorithm (RMS) and the corresponding schedulability analysis of periodic tasks. RMS is a fixed priority assignment algorithm that is performed off-line, where task priorities are assigned in relation to the frequency of the task, i.e. the shorter the task period is, the higher priority the task gets. The assumptions underpinning the RMA are the following:

A1 Tasks are periodic.

A2 Worst-case execution times are known a priori, and all instances of

[3] Cascaded triggering is not allowed, i.e. transactions triggered by "triggering transactions" are always "non-triggering transactions".

periodic task τ_i have the same worst-case execution time w_i.

A3 The deadline of a task equals its period.

A4 Tasks are independent.

Lehoczsky and Ramos-Thuel [LRT92] developed the slack stealing algorithm for fixed-priority systems to which several extensions have been suggested [RTL93, RTL94, DTB93]. Lehoczsky et al. [LRT92] showed how the slack stealing algorithm guaranteed periodic hard-critical tasks and how sporadic soft tasks were scheduled by "stealing" slack time from the periodic tasks. Thuel et al. [RTL93, RTL94] extended the model to incorporate hard-essential aperiodic tasks which were admitted to the system based on an acceptance test guaranteeing the timeliness of the task (tasks not passing the test are rejected). The assumption is that periodic tasks are guaranteed to meet their deadlines, and aperiodic requests are handled on a best-effort basis, but once aperiodic requests are accepted, their timeliness is guaranteed. Davis et al. [DTB93] extended the model to incorporate periodic and sporadic hard-deadline tasks together with soft-deadline aperiodic tasks where slack is stolen from both periodic and sporadic tasks. An approximate version for dynamic task scheduling was developed and evaluated. Their approach was proven optimal in the limited case of independent periodic tasks. However, the dynamic algorithm is applicable to more general scheduling problems including hard-deadline sporadic tasks.

Kim et al. [KS95a, Kim95] developed a strict deadline-driven transaction-processing scheme for executing transactions of different criticality where database consistency and temporal consistency are maintained. Within this framework, they categorized real-time transactions of different types into three classes where classes I and II are critical and class III is non-critical. Timeliness is enforced by adopting the deadline-monotonic scheduling algorithm [ABR$^+$93] for class I and class II transactions, and a dynamic slack-stealing-based algorithm [RTL94] for class III transactions.

What is common to these approaches (i.e., rate-monotonic, deadline-

monotonic, and slack-stealing algorithm) is the assumption that periodic tasks are critical and aperiodic tasks are non-critical. These approaches reject aperiodic tasks during transient overloads, i.e., preferential treatment is given to periodic tasks. In contrast, in our proposed overload resolution strategy all transactions are handled equally since all transactions (periodic and non-periodic) must be admitted by the admission controller. OR-ULD allows non-periodic transactions that may be critical and, hence, these cannot be rejected, but must be handled either by executing the original transaction as requested or by invoking a contingency transaction.

6.4 Bias Control / Skipping

Koren and Shasha [KS95b] define the algorithms RTO (Red Tasks Only) and BWP (Blue tasks When Possible), both being variants of EDF and RMS. They consider the scheduling problem when the workload consists of periodic tasks that can occasionally be skipped. Every task has instances where each instance of a task can be red or blue. A red task instance must complete before its deadline and a blue task instance can be aborted at any time.

Hamdaoui and Ramanathan [HR95] proposed a distance-based priority (DBP) scheme, which is appropriate for periodic and non-periodic streams having (n, m)-temporal constraints, implying that the tasks must meet m deadlines in any n invocations.

In [BB97], hard-deadline tasks are assigned (n, m)-temporal constraints. Their work focuses on attempting to increase the capacity for soft tasks by allowing skips for hard tasks. Hard tasks are periodic and soft tasks are aperiodic.

Caccamo and Buttazzo have proposed an algorithm based on EDF that is appropriate for workloads consisting of periodic firm deadline tasks allowing skips and aperiodic soft tasks [CB97]. Similar to [BB97], the idea is to enhance the responsiveness for aperiodic tasks by exploiting skips for periodic tasks.

In contrast to all the above workloads ([KS95b, HR95, CB97, BB97]), the workload studied in this thesis consists of sporadic, critical transactions that are non-skippable, and non-critical transactions with completion ratio constraints. While critical transactions cannot be skipped, they can be replaced with contingency transactions during (transient) overloads. OR-ULD/BC resolves transient overloads by using dual strategies; invoking contingency transactions that replace original transactions, and controllably dropping firm transactions considering the skip constraint. OR-ULD/BC assumes the alternative statistical model of skips suggested by Koren and Shasha [KS95b], i.e., some specified fraction of deadlines must be met during a finite time interval.

Pang et al. [PLC92] studied the behavior of EDF and AED (see page 115) when scheduling multi-class transaction workloads in real-time database systems. They noticed that both EDF and AED, in addition to the fact that their performance deteriorates rapidly when the system is overloaded, discriminates longer transactions in attempting to minimize the number of tardy transactions and, hence, are not appropriate for multi-class transaction workloads. They therefore designed Adaptive Earliest Virtual Deadline (AEVD) with the goal of having the performance of EDF but without its biased behavior. In their model, transaction classes are distinguished by the mean size of the transactions belonging to a class and the tightness of the time constraints is proportional to transaction sizes. The idea is that the average completion ratios of the transaction classes (CCR_i) should ultimately be the same. Hence, they do not handle minimum class completion ratio (MCCR) constraints.

Datta et al. developed an algorithm called AAP (Adaptive Access Parameter) [DMK+96] for disk-resident and firm real-time database systems which incorporates an admission controller that tests new transactions for schedulability. AAP performs admission control for managing transient overloads and to bias resource allocation towards particular transaction classes, which are distinguished by their mean sizes. The goal of AAP is to ensure fairness between transaction classes. AAP and AEVD are capable of handling workloads consisting of multiple transaction classes where all transactions have the same criticality. However,

AAP and AEVD are not able to handle transactions having a contingency transaction. In contrast to the work of Pang et al. and Datta et al., OR-ULD handles transactions of different criticalities, and in our approach classes are distinguished by their minimum class completion ratio (not their size). Moreover, our goal is to bias the execution towards classes having high class completion ratio requirements while ensuring that the minimum class completion ratio requirements are not violated.

6.5 Alternative Action Models

Considerable work has been carried out on the use of task/transaction models having alternative actions. Overall, the different approaches can be categorized by how a task is decomposed and the criteria for executing the decomposed subtasks, i.e., how they are dependent upon each other. In the *imprecise computation model* (section 6.5.1), decomposition of tasks into one mandatory and one optional subtask is suggested where the former computes a result which satisfies the minimum requirement. Additional execution of the optional part increases the quality of the result. Execution of both the mandatory part and the optional part will provide a result with no quality reduction or errors. Hence, during overloads only mandatory tasks are executed. In the *primary/backup model* (section 6.5.2), tasks have a primary copy and a backup copy where the backup is a copy of the primary task and has the same temporal scope as the primary task. In the *original/contingency model* (section 6.5.3), tasks have an original transaction and a contingency transaction where the contingency transaction produces satisfactory results but it is not a copy of the original transaction. The backup and the contingency are executed in order to recover the system from the failure of completing the primary, for example timing faults of primaries [LC86, CC89, Che94, Nag97], processor failures [KS86, YS92, MMG94], or database consistency faults [SKS95].

6.5.1 Imprecise Computation Model

Shih et al. [SLC89] consider how to feasibly schedule tasks consisting of a mandatory and an optional subtask. They consider the special case when optional tasks have identical values or identical processing times. In their model, developed for single-processor systems, tasks are preemptable, aperiodic, have values representing importance, arbitrary and identical ready times and deadlines, and the worst-case execution times are known. They develop a set of algorithms: DFS and F(identical processing times), LDF (identical ready times), which prove to produce feasible schedules under those conditions.

In [LLS+91] Liu, Shih et al. define a set of scheduling algorithms appropriate for imprecise computation tasks. Their approach is to guarantee mandatory subtasks by considering them critical and then on a best-effort basis schedule the optional subtasks.

In [SL92] Shih et al. propose a set of algorithms (NORA – No-Off-line tasks and on-line tasks Ready upon Arrival, ORA – On-line task Ready upon Arrival, OAR – On-line Tasks with Arbitrary Ready time) that focus on minimizing the total error (in this work the assumption about tasks having identical weights has been eliminated as compared to [SLC89]). Lee et al. [LRS+98] have proposed an extension to the OAR algorithm, making it suitable for minimizing the largest weighted error.

Kim et al. [KSCJ98] use a strict two-level queue system, where tasks in each queue are scheduled by EDF. Mandatory and optional parts of released tasks are placed in different queues. The idea is to schedule and execute mandatory parts first, and then execute optional parts. (This is performed if and only if the mandatory part of a released task has completed and no mandatory parts among the released tasks are unfinished.) Simulation results show that the algorithm exhibits similar performance with respect to maximum error (accuracy) while maintaining higher schedulability to the algorithms proposed by Shih et al. [SLC89, SL92].

In our proposed overload resolution strategy, in contrast to the impre-

cise computation model, critical transactions have one original part which produces a result of full accuracy, and one contingency transaction which produces a result of reduced quality. Hence, while it is always necessary to complete mandatory parts of imprecise transactions, original transaction and contingency transactions have a different constraint; either the original transaction or the contingency transaction is executed to completion. Hence, during overloads the contingency transactions replace the original transactions.

It should be noted that our approach is also applicable for imprecise transaction workloads. The performance of OR-ULD for imprecise transaction workloads has been evaluated and compared to the performance of the algorithm proposed by Kim et al. [KSCJ98]. Simulation results showed that OR-ULD performs significantly better during light to medium overloads and provides better overall accuracy [HT, Thu99].

6.5.2 Primary/Backup Model

Krishna and Shin [KS86] study the problem of tolerating processor failures in a multiprocessor system by executing multiple copies of each task on different processors. Tasks are periodic with known worst-case execution times.

Oh and Son [YS92] have considered the problem of handling processor failures in hard real-time multi-processor systems. Their goal is to maximize the number of processor failures to be tolerated and minimize the number of processors used. Tasks have one primary copy and one backup copy, hence, they are equal in size, etc. Oh and Son proposed a scheduling algorithm based on the FFD (First-Fit Decreasing) bin packing heuristic. Their approach is to have one primary schedule and one backup schedule where the latter is invoked in case of processor failures.

In comparison, while Krishna et al. [KS86] and Oh et al. [YS92] focus on processor failures in multi-processor systems, OR-ULD focuses on resolving transient overloads locally in a single-processor system. Hence, contingency transactions are executed on the same processor as the origi-

nal, and contingency transactions have different characteristics compared to the original transactions.

6.5.3 Original/Contingency Model

In [CC89] Chetto et al. modify EDF with a deadline mechanism appropriate for scheduling original transactions and contingency transactions in fault-tolerant real-time systems. The strategy, denoted EDL (Earliest Deadline Last chance strategy), lets the scheduler reserve time intervals for executing alternative transactions. Alternatives are scheduled for execution at their latest time (i.e. $\bar{d}_i - \bar{w}_i$). Original transactions are scheduled in remaining times before their contingency transactions and, hence, whenever an original transaction successfully completes, execution of the corresponding contingency transaction is no longer necessary. This implies that any contingency transaction may terminate a currently executing original transaction for starting its execution at the correct time. Moreover, while the worst-case execution times of contingency transactions are necessary, the execution times of original transactions do not have to be known. The EDL strategy ensures that a feasible schedule for contingency transactions will be found if there exists such a schedule. In contrast to OR-ULD, contingency transactions are used for resolving transient overloads as opposed to handling timing failures. The EDL strategy is designed specifically for workloads where all transactions have contingency transactions. Overloads are resolved by simply rejecting transactions at admission control level. OR-ULD has the advantage of sophisticated overload management. Chetto [Che94] also shows how the deadline mechanism can be extended to distributed systems in order to cope both with processor failures and timing faults.

Nagy [Nag97, BN97] studies the problem where transactions consist of a primary task and a compensating task. Admitted transactions are guaranteed to complete either by successful commitment of the primary task or by safe termination (successful commitment) of the compensating task. The worst-case execution time of the compensating task is known a priori, but not for the primary task. The compensating task is a safe

mechanism for bailing out if the primary transaction is not able to finish before its deadline due to the unknown processing requirements. Hence, compensating tasks are not used for resolving overloads. Instead, overloads are resolved at the admission control level, rejecting tasks at submission time. Successful commitment of a primary task gives a constant benefit to the system (represents the importance of the transaction) but successful completion of the compensating transaction offers no benefit to the system. In contrast, in our overload resolution approach contingency transactions are used as one mechanism for resolving transient overloads. The other mechanism is rejection of transactions at the admission control level. In other words, our proposed strategy performs overload resolution by rejecting transactions at the admission control level and re-allocating resources among admitted transactions. Hence, while admitted transactions are not guaranteed to complete, non-critical transactions may be dropped and critical transactions may be replaced by their contingency transactions but not dropped. Moreover, OR-ULD assumes that worst-case execution times of original and contingency transactions are known. Admission is based on execution time and utility (during transient overloads). In addition, completing contingency transactions contribute with some utility, although reduced with respect to the original transaction.

Soparkar et al. [SKS95] suggest a predicate-based model where deadlines are attached to contingency constraints rather than directly to transactions. In their model, validity of the constraints represents a safe database state. Violation of the constraints indicates the occurrence of a crisis, i.e., a situation calling for corrective actions to be taken. The system model comprises finite sets $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of predefined transactions, and $C = \{c_1, c_2, \ldots, c_m\}$ of predefined contingency constraints. The contingency constraints are in the form of conjuncts [SKS95]. The database is considered to be in a safe state when $\bigwedge_{i=1}^{m} c_i$ is satisfied. Their work focuses on optimal strategies for restoring the database to a state consistent with the contingency constraints, e.g. selection of contingency actions which is shown to be NP-complete.

A number of intrinsic differences can be identified. First, the suggested approach assumes that the execution of normal transactions co-exists

with the contingency actions, as opposed to OR-ULD where normal transactions are substituted with contingency actions which are executed instead. Second, corrective actions are initiated by an activation module that continuously monitors the state of the database detecting inconsistencies, resulting in a potential deterioration of performance. Our overload resolution strategy suggests that contingency actions should be activated by the real-time scheduler detecting an overload. The actual activation and detection is modeled with ECA rules. Third, their study considers only internally generated transactions and excludes transactions with external inputs or outputs.

6.6 A Tabular Overview

This chapter concludes with tables listing several criteria focusing on scheduling assumptions and models. The tables represent a summary of related research and provide an additional instrument for comparing our proposed overload resolution strategy to other strategies. The following symbols are used in the tables:

x	—	feature applies for all transactions or feature exists in the system;
c/n	—	feature applies for **c**ritical/**n**on-critical transactions; and
p/s/a	—	feature is true for **p**eriodic/**s**poradic/**a**periodic transactions.

Characteristics of Scheduling Entity		OR-ULD	[Nag97]	[DMK+96]	[LC86]	[CC89]	[BSS95]	[LLS+91]	[Loc86]	[HLC91]	[PLC92]	[SST+96]	[RTL93]	[DTB93]
A. Inter-arrival time	1) periodic	x			x	x			x			x	x	x
	2) sporadic	x				x								x
	3) aperiodic	x	x	x			x	x	x	x	x	x	x	
B. Exec. time	1) w_i known a priori	x		x[1]	x	x	x	x	x			x	x	x
C. Deadline	1) unknown in advance	sa	a	x			x	x	x		x	x	a	
	2) end of period	p			x	x							p	x
D. Deadline criticality	1) critical	ps	x[2]		x	x							p	ps
	2) non-critical	a	x[3]	x			x	x	x	x	x	x	a	s
E. Value func. of τ_i	1) importance						x	x	x	x[4]				
	2) value functions	x	x						x					

Table 6.1: Comparison of timing characteristics of scheduling entity

Algorithms		OR-ULD	[Nag97]	[DMK+96]	[LC86]	[CC89]	[BSS95]	[LLS+91]	[Loc86]	[HLC91]	[PLC92]	[SST+96]	[RTL93]	[DTB93]
Scheduling Model														
F. Number of processors	1) single	x[5]	x[5]		x	x	x	x					x	x
	2) multiple			x				x	x	x	x	x		
G. Scheduling entity	1) transaction	x	x	x						x	x	x		
	2) task				x	x	x	x	x				x	x
H. Scheduling paradigm	1) static				x								c	x
	2) dynamic	x	x	x		x	x	x	x	x	x	x	n	
I. Type of guarantee	1) absolute		x				x		x				c	c
	2) conditional	x		x		x								
	3) best effort							x		x	x	x	n	n
J. Admission controller		x	x	x		x	x						x	
K. Performance metric	1) maximize total value	x	x				x	x	x	x[4]				
	2) minimize total error							x						
	3) max. # primaries				x									
	4) completion ratio	x		x		x				x	x	x	x	
	5) normalized compl. ratio	x		x							x			
L. Overload resolution strategy	1) reject transactions	a	x	x			x						x	x
	2) invoke alt. trans.	c			x	x								
	3) run only mandatory							x						
	4) abort transactions	n								x[8]	x[8]	x[8]		x

Table 6.2: Comparison of scheduling models

Algorithms / Database Model		OR-ULD	[Nag97]	[DMK+96]	[LC86]	[CC89]	[BSS95]	[LLS+91]	[Loc86]	[HLC91]	[PLC92]	[SST+96]	[RTL93]	[DTB93]
M. Concurrency control meth.	1) optimistic	x	x	x							x			
	2) pessimistic								x			x		
N. Read/write sets	1) known													
	2) unknown	x	x	x					x	x	x			
O. Coupling modes	1) immediate											x		
	2) deferred											x		
	3) det. excl ser.	x												
	4) detached	x												
P. DB storage	1) disk			x						x	x			
	2) main memory	x	x						x					

Table 6.3: Comparison of database models

Algorithms		OR-ULD	[Nag97]	[DMK+96]	[LC86]	[CC89]	[BSS95]	[LLS+91]	[Loc86]	[HLC91]	[PLC92]	[SST+96]	[RTL93]	[DTB93]
Alternative Actions														
Q. Alternative actions	1) mandatory/optional							x						
	2) primary/backup													
	3) original/contingency	x	x		x	x								
R. Characteristics of alternative actions	1) $\bar{d}_i = d_i$	x	x		x	x		x						
	2) $\bar{d}_i > d_i$	x												
	3) $\bar{w}_i \ll w_i$ [6]	x	x[7]		x									
	4) $\bar{w}_i \cong w_i$	x	x		x									
	5) $\bar{v}_i$ supported	x						x						
	6) $\bar{w}_i$ known a priori	x	x		x	x		x						

Table 6.4: Comparison of alternative action models

The comparison focuses on the characteristics of (i) the time constraints of the scheduling entities (6.1; category A-E); (ii) the scheduling model (6.2, category F-L); (iii) the database model (6.3, category M-P); and (iv) alternative actions (6.4, category Q-R). Below follows a description of the criteria.

A. Inter-Arrival Time of Transactions Approaches are compared with respect to arrival patterns of transactions. The alternatives are:

1. periodic;
2. sporadic; and
3. aperiodic.

B. Execution Time of Transactions Compares the a priori knowledge about transaction execution times. The alternative is:

1. worst-case execution times of transactions are known a priori.

C. Deadline Compares the a priori knowledge of transaction deadlines. The alternatives are:

1. deadlines are not known a priori to transaction arrival; and
2. deadlines of periodic transactions equal end of their period.

[1]Is computed upon arrival by using read and write sets which are known.

[2]System is not designed for having transactions with **and** without deadline tolerance at the same time.

[3]System has not been evaluated for soft deadline transactions.

[4]Only for HED (Hierarchical Earliest Deadline) algorithm.

[5]One processing element is used for executing transactions, and in addition, there is a dedicated processing element for performing scheduling activities.

[6]Execution time of alternative action is significantly smaller, although not necessarily orders of magnitude smaller.

[7]System was evaluated when either 3) or 4) is true.

[8]Transactions aborted when they become tardy.

D. Deadline Criticality Comparison of supported deadline criticality of transactions. The alternatives are:

1. critical; and
2. non-critical.

E. Value Functions Comparison of support for associating value functions to transactions. The alternatives are:

1. importance is associated with transactions and does not change over time; and
2. value functions are used to represent the time-varying utility contributed to the system (includes possible penalty).

F. Number of Processors Compares the number of processing elements in the system on which application transactions are executed. The alternatives are:

1. single processor; and
2. multiple processors with shared memory (centralized system).

G. Scheduling Entity Compares the type of scheduling entity. The alternatives are:

1. task; and
2. database transaction.

H. Scheduling Comparison of *when* scheduling decisions are made. The alternatives are:

1. static (off-line; alternative includes hybrid solutions that are both static and dynamic); and
2. dynamic (on-line).

I. Type of Guarantee Compares the type of guarantee that is provided by the scheduler for a set of transactions. The alternatives are:

1. absolute guarantee, i.e. transactions are guaranteed to meet their time constraints once they passed a schedulability test;
2. conditional guarantee, i.e. transactions should meet their time constraints (once admitted), but may be subject to re-negotiation under certain conditions; and
3. best effort, i.e. transactions are not given any guarantees.

J. Admission Controller Specifies whether the system performs admission control upon arrival or activation of transactions.

K. Performance Metric The performance metric used to assess the relative merit of each approach is:

1. maximize for total value (includes maximize hit value ratio, see page 115);
2. minimize total error (includes minimize loss ratio);
3. maximize the number of completions of original transactions;
4. completion ratio; and
5. normalized completion ratio for a set of transaction classes.

L. Overload Resolution Strategy Compares the strategy used to resolve overloads. The alternatives are:

1. selected transactions are rejected upon arrival;
2. alternative transactions are invoked (alternative action is a separate transaction from the original transaction, i.e., not the mandatory/optional scenario);

3. only the mandatory part of transactions are executed during overloads and optional parts are rejected; and
4. currently executing transactions may be aborted during overloads.

M. Concurrency Control Method Specifies which type of concurrency control method is used, if any. The alternatives are:

1. optimistic concurrency control; and
2. pessimistic concurrency control.

N. Read/Write Sets Specifies whether the read/write set is known a priori or not (determines whether conflicts can be prevented or not). The alternatives are:

1. known; and
2. unknown.

O. Coupling Modes Compares the supported coupling modes (see section 2.5.2 for a description of the coupling modes). The alternatives are:

1. immediate;
2. deferred;
3. detached exclusive serial; and
4. detached.

P. Database Storage Compares the type of data storage used for storing data objects. The alternatives are:

1. disk storage; and
2. main memory.

Q. Alternative Action Compares the type of alternative action model supported. The alternatives are:

1. mandatory/optional (imprecise transactions);
2. primary/backup; and
3. original/contingency action.

R. Characteristics of Alternative Actions Compares how alternative actions are related to the original transaction with respect to certain attributes. The alternatives are:

1. alternative transaction has the same deadline as the original transaction;
2. alternative transaction may have a deadline that is later than the original transaction;
3. the worst-case execution time of the alternative transaction is significantly smaller than the worst-case execution time of the original transaction (not necessarily orders of magnitude but significantly);
4. the worst-case execution time of the alternative transaction and the original transaction is similar (however, the worst-case execution time of the alternative transaction is smaller or equal to the worst-case execution time for the original transaction);
5. alternative transactions have value functions; and
6. worst-case execution times of contingency transactions are known a priori.

Chapter 7

Conclusions

We shall not cease from exploration
and the end of all our exploring
will be to arrive where we started
and know the place for the first time.
- T.S. Eliot

This final chapter starts with a summary of our work (section 7.1). Research contributions are presented (section 7.2), followed by a discussion of the research results and the applicability of the approach to other workload scenarios (section 7.3). In the last section (section 7.4) we identify issues for future work.

7.1 Summary

Dynamic real-time systems offer both flexibility and adaptability in handling new situations occurring in the environments in which they are working, and dynamic real-time scheduling algorithms generally achieve good performance during light to moderate workloads. However, dy-

namic real-time systems are prone to transient overloads, i.e., a situation lasting for a finite interval where system resources are saturated and, hence, not all system requests can be handled. The primary responsibility of a dynamic scheduler in an under-utilized system is to determine in which order to execute transactions such that their time constraints are enforced. In the case of a transient overload, however, not all transactions can meet their time constraints based on their original resource requirements. In order to optimize usage of system resources it is imperative that they are granted to transactions that have critical time constraints. This implies that the scheduler has to select which transactions should be allowed to run, in addition to deciding the transaction execution order.

In this thesis the research problem of how to dynamically resolve transient overloads in dynamic real-time systems in general, and in real-time database systems with complex workloads in particular, has been investigated. Specifically, the workload consists of multiple transaction classes where transactions have mixed criticality (critical and non-critical deadlines). Moreover, critical transactions are periodic and/or sporadic, and non-critical transactions may also be aperiodic. Further, critical transactions have contingency transactions, having significantly less computational requirements. Contingency transactions produce results that are of reduced quality but still acceptable. A contingency transaction can be invoked during transient overloads, replacing an original transaction, releasing some resources. Value functions are used to represent transaction importance and transaction criticality. Value functions represent the benefit (positive utility) contributed to the system upon successful transaction completion, and the penalty (negative utility) imposed on the system if a transaction becomes tardy. Consequently, for contingency transactions, the value function also represents the degree of quality reduction.

7.2 Contributions

The aim of this work has been to investigate how dynamic real-time systems can be made overload tolerant and how predictable behavior can be enforced during transient overloads. Our work complements current research on scheduling and overload management in dynamic real-time systems and particularly in real-time database systems. We propose a framework that exploits dynamic overload resolution and enables real-time systems to dynamically handle extreme transaction workloads of mixed criticality. More specifically, the contributions of our framework are fourfold: Overload resolution follows (i) a novel strategy where a set of overload resolution actions is generated. Overloads are resolved by carefully selecting a set of overload resolution actions that release the necessary amount of time among currently executing transactions in order to admit a new transaction. We have introduced (ii) a scheduler architecture that supports admission control and dynamic overload resolution. Moreover, (iii) a dynamic and utility-driven overload resolution algorithm (OR-ULD) that implements the strategy has been developed. OR-ULD gracefully degrades performance during transient overloads and still ensures timeliness of critical transactions. OR-ULD has been equipped with (iv) a novel bias control mechanism (/BC) which during transient overloads increases the bias as the class completion ratio decreases towards a specified minimum level.

7.2.1 Overload Resolution Strategy

In the case of transient overload and a new transaction arriving to the system, there is a required amount of time that must be released if the new transaction is to be admitted. Our strategy scrutinizes resource reservations of already admitted transactions and attempts to release resources among these transactions (e.g., dropping a transaction, replacing a transaction, or postponing a transaction). The strategy releases sufficient resources among admitted transactions to admit the new transaction. In doing so, it considers the utility loss imposed by releasing

resources and the utility gain of admitting the new transaction to the system as opposed to the penalty imposed on the system if it were rejected.

7.2.2 Scheduling Architecture

The proposed scheduling architecture incorporates an admission controller, a scheduler, an overload resolver, and a dispatcher. The purpose of admission control is to prevent the system from being saturated. The admission controller tests new transactions for schedulability upon their arrival and only transactions passing the schedulability test are granted system resources. Hence, the admission controller presents only schedulable transaction workloads to the scheduler and thereby the admission controller constitutes an overload filter. There is a disadvantage of applying admission control as the single mechanism for handling transient overloads for multi-class transaction workloads. There are two main approaches when using admission control for multi-class transaction workloads. Either (i) divide and assign system resources per transaction class and perform admission control for each transaction class separately (i.e., multiple admission controllers are used), or (ii) use a single admission controller for all transaction classes (i.e., admission control is independent of transaction class). The former approach suffers from under-utilization caused by the fact that system resources are pre-assigned to classes, and thereby also a degree of limited flexibility. Hence, new transactions may be rejected due to scarce resources within a class while the overall system may be under-utilized. The second approach, which seems more promising, generates higher resource utilization and offers a higher degree of flexibility. The primary concern with this approach is related to the nature of transaction classes, particularly how classes are related with respect to importance and criticality. Since admission is resource based and does not consider importance and criticality, critical transactions may be rejected due to earlier admissions and, in the worst case, earlier admissions of non-critical transactions. Hence, in order to enforce admission of a newly arrived critical transaction earlier admissions have

to be scrutinized and re-evaluated, and if necessary, resources may have to be released in order to admit the new transaction.

7.2.3 OR-ULD Overload Resolution Algorithm

The OR-ULD algorithm has been implemented and an extensive simulation-based performance analysis has been conducted. The results show that the OR-ULD algorithm (i) gracefully degrades performance during overloads by increasing the number of contingency transactions (replacing the original transactions) and gradually dropping non-critical transactions, (ii) ensures the timeliness of critical transactions (within a certain operational envelope), and (iii) produces near-optimal results. A system utilizing both admission control and overload resolution by resource re-allocation, as outlined in this thesis, offers a substantial increase in completion ratio during transient overloads. In particular, the enforcement of critical time constraints is guaranteed in a wider operational envelope.

In our work, transactions are scheduled using EDF and blocking is handled by the Stack Resource Policy (SRP) [Bak91]. Admission considers the schedulability of a transaction given its worst-case execution time and the potential time it may be blocked.

7.2.4 Bias Control

It is generally accepted that occasionally missing some firm and soft deadlines is acceptable and is not considered a failure. Hence, during transient overloads, some of the non-critical transactions can be dropped without jeopardizing system correctness. Most research in the area of scheduling soft and firm transactions assumes that there is no requirement on the minimum number of firm or soft transactions that must successfully complete. The focus of this thesis has been on scheduling multi-class transaction workloads where transaction classes are distinguished by their minimum class completion ratio requirement. System correctness is maintained if, for all transaction classes, the class comple-

tion ratio does not fall below the minimum level.

Even though OR-ULD ensures enforcement of critical time constraints, while doing a best effort at executing non-critical transactions, it has no way of meeting specific completion ratio requirements for the non-critical transactions. The results show that OR-ULD/BC (i) gradually drops non-critical transactions as the load increases towards the operational envelope, and (ii) the suggested bias control mechanism enforces transaction class completion ratio requirements within this operational envelope.

7.2.5 Advantages of Our Approach

The approach, encompassing the listed contributions, is unique and has several advantages compared to other approaches. First, the overload resolution strategy takes advantage of the expressive power of value functions. Value functions represent the positive utility (benefit) contributed to the system if the transaction completes on time, and the possibly negative utility (penalty) imposed in case the deadline is missed. The natural and explicit representation of criticality using negative utility enables OR-ULD to enforce time constraints of critical transactions without using artificial values or constructs. Second, the approach separates overload management from scheduling and admission control. The overload management can be used with other admission control and scheduling policies. The scheduling policies are required to have a mechanism for detecting transient overloads and must be able to indicate the critical interval in which a transient overload is occurring. Third, the approach enables resolution of transient overloads by using multiple strategies, i.e., dropping transactions, replacing transactions with contingency transactions, and/or deferring completion of transactions till after their deadlines but within their deadline tolerance. For the first time it is possible to balance these decisions against each other. This is done by analyzing the consequences of an overload resolution action with respect to utility. This means that for a transient overload situation one can determine whether it is more beneficial to invoke contingency transactions of crit-

ical transactions or to drop non-critical transactions in order to be able to admit a new critical transaction.

7.3 Discussion

In contrast to the imprecise computation model [LLS+91], we have focused on how and when to *replace* critical transactions, as opposed to only *partially executing* critical transactions, and particularly on the case when critical transactions have contingency transactions where the original transaction or the contingency transaction is executed to completion. In this case, the criticality of a contingency transaction is given by the criticality of the original transaction. In contrast, the imprecise computation model suggests decomposition of critical transactions into mandatory parts and optional parts, having hard deadlines and firm deadlines respectively. Moreover, only mandatory transactions are executed during overloads while optional transactions are dropped [SLC89, SL92, LLS+91].

Imprecise transactions can be incorporated to our scheduling model by defining mandatory parts to be hard critical transactions without contingency transactions. Further, optional parts are defined to be non-critical transactions with firm deadlines. Since the optional part should not execute before the corresponding mandatory part has completed, the imprecise computation model has a precedence constraint between mandatory and optional parts. While our scheduling model does not explicitly support the representation of precedence constraints, this causal chain can be modeled by letting mandatory transactions, once they complete, release their optional transactions. Current work focuses on extending OR-ULD to include imprecise transactions [HS99, Thu99].

We have studied the case when all critical transactions are assumed to have a contingency transaction but the OR-ULD algorithm can be applied to transaction workloads where only a subset of the critical transactions have contingency transactions. Let us explain why. Consider critical transactions having contingency transactions to be semi-critical

and their contingency transactions critical. Further, critical transactions without contingency transactions are truly critical since they must complete and cannot be dropped or replaced. If the set of critical transactions is schedulable, excluding semi-critical transactions, the OR-ULD algorithm still works. When generating overload resolution actions, any action suggesting that a critical transaction should be dropped imposes infinite utility loss, implying that it will not be selected as an appropriate overload resolution action. As it turns out, this is what happens during heavy overloads where a dominant part of the critical transactions has been replaced with their contingency transactions. If a new transaction arrives to the system, as our performance analysis shows, admitted contingency transactions remain unchanged and they are not dropped.

Our performance analysis has primarily focused on workloads where only critical transactions have contingency transactions. However, our strategy and the OR-ULD algorithm are also applicable to more generalized workloads, where all or some of the non-critical transactions may have contingency transactions as well. Hence, a non-critical transaction can be rejected, dropped or replaced with its contingency transaction. Further, contingency transactions of non-critical transactions can be dropped as opposed to contingency transactions of critical transactions. The effect of this is that the ratio of successfully completed non-critical transactions is likely to increase during light overloads. In the case where an overload increases in severity, there is a potential risk of an additional increase of resources wasted due to dropping partially executed contingency transactions of non-critical transactions.

7.4 Future Work

We have evaluated the performance of OR-ULD for workloads consisting of critical and non-critical transactions with no deadline tolerance. Hence, the two types of overload resolution actions used are dropping and replacing a transaction. However, there is a third type of overload resolution action, namely postponing a transaction outside the critical

overload interval. Postponing a transaction till after its deadline reduces the benefit contributed to the system. However, postponing a transaction is a more complex overload resolution action than dropping transactions or replacing transactions with contingency transactions, since it will affect the execution order of admitted transactions. An analysis is required in order to evaluate the efficiency if this type of action.

The work in this thesis has focused on transactions where the utility contributed to the system upon successful completion is constant as long it completes before the deadline. The expressive power of value functions enables representation of importance for transactions where the utility contributed to the system may change over time (see section 3.2.1 for a description of value functions). In order to optimize the total utility contributed to the system, transactions contributing with the highest utility should be admitted. In addition, these transactions should be scheduled such that the utility contributed to the system is maximized. In our systems, admitted transactions are scheduled using EDF which does not consider the utility of the transactions. The utility is considered when transactions are admitted and when overloads are resolved. By extending the scheduling architecture with a component that improves the schedule outlined by EDF with respect to the utility of the transactions, altering the execution order of admitted transactions, overall utility can be maximized. It should be noted that altering the execution order of admitted transactions is also justified in systems where transactions have constant value functions. Notably, completing high-valued transactions as early as possible increases the total utility obtained by the system during transient overloads. The effect is due to the fact that when an overload occurs, reallocation of resources is among transactions that contribute less utility.

Our approach resolves overloads by releasing time among admitted transactions. Each overload resolution action saves some time but results in a utility loss, representing the fact that the quality of the result produced by a transaction is traded for time. Currently transactions are only downgraded and never upgraded again, i.e., dropped transactions are not considered for re-invocation, and replaced transactions are not

substituted with their original transactions. This is particularly justified in systems where the difference between worst-case execution times and actual execution times of the transactions is small and can be considered negligible in comparison to the cost associated with upgrading transactions again. For example, in main-memory resident systems blocking is reduced, or even eliminated, due to negligible I/O delays. However, performing admission and scheduling of transactions based on their worst-case execution times may be too pessimistic in systems where actual execution time is significantly smaller than the worst-case execution time. This causes the system to be not fully utilized, in which case it could be beneficial to re-invoke dropped transactions and replace contingency transactions with their original transactions.

References

[ABR+93] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[ABW93] N. C. Audsley, A. Burns, and A. J. Wellings. Deadline monotonic scheduling theory and application. *Control Engineering Practice*, 1(1):71–78, February 1993.

[AGM88] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, 1988.

[AH98] S. F. Andler and J. Hansson, editors. *Proceedings of the Second International International Workshop on Active, Real-Time, and Temporal Databases (ARTDB-97)*, number 1553 in Lecture Notes Series (LNCS). Springer-Verlag, December 1998.

[AHE+96] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS towards a distributed and active real-time database system. *SIGMOD Record*, 25(1):38–40, March 1996.

[Bak91] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems Journal*, 3(1):67–99, March 1991.

[BB95] H. Branding and A. P. Buchmann. On providing soft and hard real-time capabilities in an active DBMS. In Berndtsson and Hansson [BH95b], pages 158–169.

[BB97] G. Bernat and A. Burns. Combining (n, m)-hard deadlines and dual priority scheduling. In *Proceedings of the 18th Real-Time Systems Symposium*, pages 46–57. IEEE Computer Society, San Francisco, CA, December 1997.

[BBKZ94] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an open system: The REACH rule system. In N. W. Paton and M. H. Williams, editors, *Rules in Database Systems*, pages 111–126. Springer-Verlag, 1994.

[BH95a] M. Berndtsson and J. Hansson. Issues in active real-time databases. [BH95b], pages 142–157.

[BH95b] M. Berndtsson and J. Hansson, editors. *Proceedings of the First International Workshop on Active and Real-Time Databases (ARTDB-95)*, Workshops in Computing. Springer-Verlag, June 1995.

[BKM+91] S. Baruah, G. Koren, B. Mishra, A. Ragunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 100–110, 1991.

[BKN+98] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, and C. Weems. The Spring scheduling co-processor: A scheduling accelerator. *IEEE Transactions on VLSI*, 1998.

[BLS97] A. Bestavros, K-J. Lin, and S. H. Son, editors. *Real-Time Database Systems - Issues and Applications.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1997.

[BN92] B. Brachman and G. Neufeld. TDBM: A DBM library with atomic transactions, June 1992.

[BN96] A. Bestavros and S. Nagy. Value-cognizant admission control for RTDB systems. In *Proceedings of the 17th Real-Time Systems Symposium*, pages 230–239. IEEE Computer Society, Washington, DC, December 1996.

[BN97] A. Bestavros and S. Nagy. Value-cognizant admission control strategies for real-time dbms. In Bestavros et al. [BLS97], pages 54–61.

[BP98] A. Burns and D. Prasad. Value-based scheduling of flexible real-time systems for intelligent autonomous vehicle control. In *Proceedings of the 3rd IFAC Symposium on Intelligent Autonomous Vehicles*, pages 127–132, March 1998.

[BPB+99] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The meaning and role of value in scheduling flexible real-time systems. *to appear in Journal of Systems Architecture, Special Issue on Real-Time Systems*, 1999.

[BS93] G. C. Buttazzo and J. A. Stankovic. RED: A robust earliest deadline scheduling algorithm. In *Proceedings of Third International Workshop on Responsive Computing Systems*, 1993.

[BS95] G. C. Buttazzo and J. A. Stankovic. Adding robustness in dynamic preemptive scheduling. In D. S. Fussel and M. Malek, editors, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*. Kluwer Academic Publishers, 1995.

[BSR88] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of Real-Time Systems Symposium - Silver Spring*, pages 152–160. IEEE Computer Society Press, 1988.

[BSS95] G. C. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of Real-Time Systems Symposium*, pages 90–99. IEEE Computer Society Press, December 1995.

[Buc94] A. P. Buchmann. Active object systems. In A. Dogac, M. T. Ozsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, pages 201–224. Springer-Verlag, 1994.

[Bur91] A. Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[But97] G. C. Buttazzo. *Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications.* The Kluwer International Series in Engineering and Computer Science - Real-Time Systems. Kluwer Academic Publishers, 1997.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages.* Addison-Wesley, 1990.

[BW97a] A. Bestavros and V-F. Wolfe, editors. *Real-Time Database Systems - Research Advances.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1997.

[BW97b] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, 1997. 2nd edition.

[CB97] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *Proceedings of the 18th Real-Time Systems Symposium*, pages 330–339. IEEE Computer Society, San Francisco, CA, December 1997.

[CBB+89] S. Chakravarthy, B. Blaustein, A. P. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, M. Livny,

D. McCarthy, R. McKee, and A. Rosenthal. HiPAC: A research project in active time-constrained database management – Final technical report. Technical Report XAIT-89-02, Reference Number 187, Xerox Advanced Information Technology, July 1989.

[CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[Che94] H. Chetto. Guaranteed deadlines with dynamic recovery blocks in distributed systems. In *Proceedings of the Sixth Eureomicro Workshop on Real-Time Systems*, pages 199–204, 1994.

[CL90] M. Chen and K-J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems Journal*, 2, 1990.

[CL91] M. Chen and K-J. Lin. A priority ceiling protocol for multiple-instance resources. In *Proceedings of the Real-Time Systems Symposium*, 1991.

[CM93] S. Chakravarthy and D. Mishra. An event specification language (Snoop) for active databases and its detection. Technical Report UF-CIS-93-007, Computer and Information Sciences Dept, University of Florida, September 1993.

[CM96] K. Chen and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Real-Time Systems Journal*, 10(3):293–312, 1996.

[CRS+92] E. Casais, M. Ranft, B. Schiefer, D. Theboald, and W. Zimmer. OBST – An overview. Technical Report FZI.039.1, Forschungszentrum Informatik (FZI), June 1992.

[Der74] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of the IFIP Congress*, pages 807–813. North Holland Publishing Company, 1974.

[DLSB82] V. A. Dyck, J. D. Lawson, J. A. Smith, and R. J Beach. *Computing – An Introduction to Structured Problem Solving Using PASCAL.* Reston Publishing Company Inc., A Prentice Hall Company, 1982.

[DMK+96] A. Datta, S. Mukherjee, P. Konana, I. R. Viguier, and A. Bajaj. Multiclass transaction scheduling and overload management in firm real-time database systems. *Information Systems*, 21(1):29–54, 1996.

[DTB93] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of Real-Time Systems Symposium*, pages 222–231. IEEE Computer Society Press, December 1993.

[Eri97] J. Eriksson. Real-time and active databases: A survey. In Andler and Hansson [AH98], pages 1–23.

[Eri98] J. Eriksson. Specifying and managing rules in an active real-time database system, December 1998. Licentiate Thesis, Department of Computer Science, University of Linköping, Sweden.

[GBLR96] A. Geppert, M. Berndtsson, D. Lieuwen, and C. Roncancio. Performance evaluation of object-oriented active database management systems using the BEAST benchmark. Technical Report 96.07, University of Zurich, 1996.

[GLLRK79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling theory: A survey. *Annals of Discrete Mathematics*, 5, 1979.

[Gra92] M. H. Graham. Issues in real-time data management. *Real-Time Systems Journal*, 4:185–202, 1992.

[HA99] J. Hansson and S. F. Andler. *System Framework for Active Real-Time Database Systems*, chapter 13. In Lam and Kuo [LK99], 1999. To appear.

[Han] J. Hansson. RADEx++. Department of Computer Science, University of Skövde, Sweden, 1998.

[Har84] T. Harder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2), 1984.

[HB98] J. Hansson and M. Berndtsson. *Active Real-Time Database Systems*, chapter 18. In Paton [Pat98], 1998.

[HCL93] J. R. Haritsa, M. J. Carey, and M. Livny. Value-based scheduling in real-time database systems. *VLDB journal*, 2(2):117–152, 1993.

[HLC91] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *Proceedings of the Real-Time Systems Symposium*, pages 232–242. IEEE Computer Society Press, 1991.

[HR95] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995.

[HS99] J. Hansson and S. H. Son. *Overload Management in Real-Time Database Systems*, chapter 11. In Lam and Kuo [LK99], 1999. To appear.

[HSRT89] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time transaction processing. In *Proceedings of the 10th Real-Time Systems Symposium*. IEEE Computer Society, 1989.

[HSRT91] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991.

[HSZ99] J. Hansson, R. M. Steinsen, and T. Ziemke. Predicting transient overloads in dynamic real-time systems using artificial neural networks (tentative title), 1999. In preparation.

[HT] J. Hansson and M. Thuresson. Imprecise transaction scheduling with OR-ULD. In preparation.

[Hua91] J. Huang. *Real-Time Transaction Processing: Design, Implementation, and Performance Evaluation.* PhD thesis, Department of Electrical and Computer Engineering, University of Massachusetts, May 1991. TR-91-41.

[JLT86] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings Real-Time Systems Symposium*, pages 112–122. IEEE Computer Society, December 1986.

[KGM92] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In *The Proceedings of NATO Advanced Study Institute on Real-Time Computing.* Springer-Verlag, October 1992.

[Kim95] Y-K. Kim. *Predictability and Consistency in Real-Time Transaction Processing.* PhD thesis, School of the Engineering and Applied Science, University of Virginia, May 1995.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Longman, 3rd edition, 1998.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[KRP+93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practioners's Handbook for Real-Time Analysis.* Kluwer Academic Press, 1993.

[KS86] C. M. Krishna and K. G. Shin. On scheduling tasks with a quick recovery from failure. *IEEE Transactions on Computers*, C-35(5):448–455, May 1986.

[KS95a] Y-K. Kim and S. H. Son. *Predictability and Consistency in Real-Time Database Systems*, chapter 21, pages 509–531. In Son [Son95], 1995.

[KS95b] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 15th Real-Time Systems Symposium*, pages 110–117. IEEE Computer Society Press, December 1995.

[KSCJ98] J-h. Kim, K. Song, K. Choi, and G. Jung. Performance evaluation of on-line scheduling algorithms for imprecise computation. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, pages 217–222. IEEE Computer Society Press, 1998.

[KV93] H. Kopetz and P. Verissimo. *Design of Distributed Real-Time Systems*, chapter 16. In Mullender [Mul93], 1993.

[Law92] H. W. Lawson. *Parallel Processing in Industrial Real-Time Applications*. Prentice-Hall, Inc., 1992.

[LC86] A. Liestman and R. Campbell. A fault-tolerant scheduling problem. *IEEE Transactions on Software Engineering*, 12(11):1089–1095, November 1986.

[LK99] K-y. Lam and T-w. Kuo, editors. *Real-Time Database Systems*. Artech House, 1999. To appear.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20:46–61, January 1973.

[LL91] G. Le Lann. Designing real-time dependable distributed systems. Technical Report 1425, INRIA, France, April 1991.

[LLS+91] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. In Andre' van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing - Scheduling and Resource Management*, chapter 8. Kluwer Academic Publishers, 1991.

[Loc86] C. D. Locke. *Best-effort decision making for real-time scheduling.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, May 1986. Technical Report CMU-CS-86-134.

[Loc96] D. Locke. *Real-time databases: Real-world requirements*, pages 83–91. In Bestavros et al. [BLS97], 1996.

[LRS+98] C. Lee, W. Ryu, K. Song, G. Jung, and S. Park. On-line scheduling algorithms for reducing the largest weighted error incurred by imprecise tasks. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, pages 137–144. IEEE Computer Society Press, 1998.

[LRT92] J. P. Lehoczsky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the Real-Time Systems Symposium*, pages 110–123. IEEE Computer Society Press, Los Alamitos, California, December 2-4 1992.

[LSS87] J. P. Lehoczsky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE Computer Society Press, December 1987.

[Lun97] J. Lundström. A conflict detection and resolution mechanism for bounded-delay replication. Master's thesis, Report number HS-IDA-MD-97-10, Department of Computer Science, University of Skövde, Sweden, 1997.

[MD78] A. K. Mok and M. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of The IEEE Texas Conference on Computing Systems*, November 1978.

[Mel] J. Mellin. Predictable event monitoring. Licentiate Thesis, Department of Computer Science, University of Linköping, Sweden, 1998.

[MG87] D. Mandrioli and C. Ghezzi. *Theoretical Foundations of Computer Science.* John Wiley & Sons, Inc, 1987.

[MHA97] J. Mellin, J. Hansson, and S. F. Andler. Refining timing constraints of applications in DeeDS. In Bestavros et al. [BLS97], chapter 18, pages 325–343.

[MMG94] D. Mosse, R. Melhem, and S. Ghosh. Analysis of fault-tolerant multiprocessor scheduling algorithm. In *IEEE Fault Tolerant Computing*, pages 16–25, 1994.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT PRESS, 1985.

[MS99] R. Mar Steinsen. Predicting transient overloads in real-time systems using artificial neural networks. Master's thesis, Department of Computer Science, University of Skövde, 1999.

[Mul93] S. Mullender. *Distributed Systems.* Addison-Wesley, 1993.

[Nag97] S. C. Nagy. *Admission Control and Scheduling Strategies for Real-Time Database Systems.* PhD thesis, Department of Computer Science, Boston University, 1997. (http://www.cs.bu.edu/students/alumni/nagy/thesis.ps).

[OgS95] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 4(7):513–532, August 1995.

[Pat98] N. W. Paton. *Active Database Systems.* Springer-Verlag, New York, 1998.

[Pin95] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, 1995. ISBN 0-13-706757-7.

[PLC92] H. Pang, M. Livny, and M. J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of the Real-Time Systems Symposium*, pages 23–34. IEEE Computer Society Press, Los Alamitos, California, December 1992.

[Pre97] R. S. Pressman. *Software Engineering - A Practitioner's Approach.* McGraw-Hill, 1997.

[Ram93] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2):199–226, 1993.

[Ram95] K. Ramamritham. The origin of TCs. In Berndtsson and Hansson [BH95b], pages 50–81.

[Ram96] K. Ramamritham. Where do time constraints come from and where do they go? *International Journal of Database Management*, 1996.

[Raw91] G. J. E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms.* Computer Science Press, 1991.

[RSZ87] K. Ramamritham, J. A. Stankovic, and W. Zhao. Meta-level control in distributed real-time systems. In *Proceedings of IEEE 1987*, pages 10–17, 1987.

[rtd88] Special issue of real-time data-base systems, SIGMOD Record, 17:1. 1988.

[rtd96] Information systems journal: Special issue of real-time database systems, 21:1. 1996.

[RTL93] S. Ramos-Thuel and J. P. Lehoczsky. On-line scheduling of hard deadline aperiodic tasks in fixed priority systems. In *Proceedings of Real-Time Systems Symposium*, pages 160–171. IEEE Computer Society Press, Los Alamitos, California, December 1993.

[RTL94] S. Ramos-Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority scheduling using slack stealing. In *Proceedings of Real-Time Systems Symposium*. IEEE Computer Society Press, December 1994.

[SBS95] M. Spuri, G. C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proceedings Real-Time Systems Symposium*, pages 210–219. IEEE Computer Society Press, December 1995.

[SKG91] L. Sha, M. H. Klein, and J. B. Goodenough. Rate monotonic analysis for real-time systems. In Andre' van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing - Scheduling and Resource Management*, chapter 5. Kluwer Academic Publishers, 1991.

[SKS95] N. Soparkar, H. F. Korth, and A. Silberschatz. Databases with deadline and contingency constraints. *IEEE Transactions on Knowledge and Data Engineering*, 7(4), August 1995.

[SL92] W-K. Shih and J. W. S. Liu. On-line scheduling of imprecise computations to minimize error. In *Proceedings of the Real-Time Systems Symposium*, pages 280–289. IEEE Computer Society Press, Los Alamitos, California, December 2-4 1992.

[SLC89] W-K. Shih, J. W. S. Liu, and J-Y. Chung. Fast algorithms for scheduling imprecise computations. In *Proceedings of the*

Real-Time Systems Symposium, pages 12–19. IEEE Computer Society Press, 1989.

[SLC91] W-K. Shih, J. W-S. Liu, and J-Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal on Computing*, 20:537–552, 1991.

[Son95] S. H. Son, editor. *Advances in Real-Time Systems.* Prentice Hall, 1995.

[SP94] R. Sivasankaran and B. Purimetla. Design of RADEx – a real-time active database experimental system. Technical report, Department of Computer Science, University of Massachusetts, Amherst, 1994.

[SR90] J. A. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Real-Time Systems Journal*, 4(2):247–254, November 1990.

[SRC85] John A. Stankovic, Krithivasan Ramamritham, and Shengchang Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers*, C-34(12), December 1985.

[SRL87] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, Carnegie Mellon University, USA, 1987.

[SRL90] L. Sha, R. Rajkumar, and C. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SSDNB95] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling for real-time systems. *Computer*, 28(6):16–25, June 1995.

[SSH99] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time database systems. *IEEE Computer*, 32(6):29–36, June 1999.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1, 1989.

[SSRB98] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline scheduling for real-time systems – EDF and related algorithms.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1998. ISBN 0-7923-8269-2.

[SST+96] R. Sivasankaran, J. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham. Priority assignment in real-time active databases. *VLDB Journal*, 5(1):19–34, January 1996.

[Sta88a] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer 21(10)*, 21(10), October 1988.

[Sta88b] J. A. Stankovic. Real-time computing systems: The next generation. In John A. Stankovic, editor, *Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[Sta95] J. A. Stankovic. The many faces of multi-level real-time scheduling. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA'95)*, October 1995.

[TCY97] S-M. Tseng, Y. H. Chin, and W-P. Yang. Scheduling value-based transactions in real-time main-memory databases. In Bestavros et al. [BLS97], pages 115–125.

[Thu99] M. Thuresson. Imprecise computation - a real-time scheduling problem. B.Sc. dissertation, Report Number HS-IDA-EA-99-124, Department of Computer Science, University of Skövde, Sweden, 1999.

[Wei99] M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 2nd edition, 1999.

[XP90] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.

[XP93] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.

[You82] S. J. Young. *Real-Time Languages: Design and Development*. Chichester: Ellis Horwood, 1982.

[YS92] Oh Y. and S. H. Son. An algorithm for real-time fault-tolerance scheduling in multi-processor systems. In *Proceedings of the Fourth Euromicro Workshop on Real-Time Systems*, 1992.

[YWLS94] P. S. Yu, K-L. We, K-J. Lin, and S. H. Son. On real-time databases: Concurrency control and scheduling. In *Proceedings of IEEE, Special Issue on Real-Time Systems*, pages 140–157, January 1994.

[ZLLA95] W. Zhao, C. C. Lim, J. W. S. Liu, and P. D. Alexander. *Overload management by imprecise computation*, chapter 1, pages 1–22. The Kluwer International Series in Engineering and Computer Science - Real-Time Systems. Kluwer Academic Publishers, 1995.

Appendix A

Abbreviations

AAP	Adaptive Access Parameter
AED	Adaptive Earliest Deadline
AEVD	Adaptive Earliest Virtual Deadline
ATT	Admitted Transaction Table
BWP	Blue tasks When Possible
COI	Critical Overload Interval
DFS	Depth First Search
EDF	Earliest Deadline First
EDL	Earliest Deadline Last chance strategy
FFD	First-Fit Decreasing
HDF	Highest Density First
HED	Hierarchical Earliest Deadline
HVF	Highest Value First
HRF	Highest Reward First
Locke's BE	Locke's Best Effort algorithm
LDF	Latest Deadline First
LPF	Longest Processing time First
LS	Least Slack
LWF	Largest Weight First
NORA	No-Off-line tasks and on-line tasks Ready upon Arrival

OAR	On-line Tasks with Arbitrary Ready time
OCCL	Optimistic Concurrency Control using Locking
OCCL-SVW	Optimistic Concurrency Control using Locking and Serial Validation Write
OR-ULD	Overload Resolution using Utility Loss Density
OR-ULD/BC	OR-ULD with Bias Control
ORA	On-line task Ready upon Arrival
ORP	Overload Resolution Plan
RED	Robust Earliest Deadline
RMS	Rate-Monotonic Scheduling Algorithm
RTO	Red Tasks Only
SRP	Stack Resource Policy
TOI	Total Overload Interval
VID	Value-Inflated Deadline
VIRD	Value-Inflated Relative Deadline

Appendix B

Variable Descriptions

Variable	*Description*
r_i	ready time of transaction τ_i
d_i	deadline of transaction τ_i
κ_i	deadline criticality of transaction class $\mathcal{T}_i$
δ_i	deadline tolerance of transaction τ_i
t_i	termination time of transaction τ_i
w_i	worst-case execution time of transaction τ_i
ζ_i	remaining execution time of transaction τ_i
	inter-arrival time
$v_i(t)$	value function for transaction τ_i
σ_i	amount of execution time used so far for τ_i
o_i	maximum amount of time needed to abort transaction τ_i
γ_i^x	amount of utility loss due to executing resolution action x
	on transaction τ_i
ξ_i^x	amount of time saved if the resolution action x is executed
	on transaction τ_i
ρ	bias parameter

Appendix C

RADEx++ Transaction Class Description

RADEx++ [Han] is based on RADEx [SP94]. The features of RADEx have been maintained in RADEx++. In addition, RADEx++ features support generating workloads consisting of multiple transaction classes, and support value-driven scheduling and overload management.

- $NumMCTransClasses$: The number of standard transaction classes.

C.1 Original Transaction Class Description

C.1.1 General Class Description Variables

- $MCTransClassPerc[i]$: Specifies the percentage of transactions of class i. Valid values are within the range 0.0 and 100.0.
- $MCTransClassPA_B[i]$: Specifies if the transaction class should be pre-analyzed. Valid values are `TRUE` and `FALSE`.
- $MCTransCriticality[i]$: Specifies the criticality of transactions of class i. Used only with deadline-driven scheduling. Valid values are `HARD CRITICAL, HARD ESSENTIAL, FIRM, SOFT` and `OTHER`.

- $MCTransCTrans_B[i]$: Boolean value specifying whether transactions of class i should have a corresponding contingency transaction.

- $MCTransCTransType[i]$: Transactions of class i have a corresponding contingency transaction of type j (class id). Valid values are (0, ..., $NumCTransClass$).

C.1.2 Value Function Variables

- $MCTransVFType\{x\}Perc[i]$: The percentage of transactions of class i having a value function of type x ($x = 1, 2, 3$; Type 1: Constant value function; Type 2: Linearly decreasing value function (ASAP); Type 3: Linearly increasing value function (ALAP). Valid values range between 0.0 and 100.0.

- $MCTransMinVF[i]$: Minimum utility for transactions of class i. Valid values are (0, ..., 10000)

- $MCTransMaxVF[i]$: Maximum utility for transactions of class i. Valid values are (0, ..., 10000)

- $MCTransVFDist[i]$: Specifies the distribution function used for calculating the values for class i. Valid distribution are `UNIFORM` and `NORMAL`.

- $MCTransVFFact[i]$: 'Value-function factor' determines the slope of the value function by setting an interval Δu within which the lower and upper values may vary (see figure C.1).

- $MCTransMinPenalty[i]$: Minimum penalty for transactions of class i.

- $MCTransMaxPenalty[i]$: Maximum penalty for transactions of class i.

- $MCTransPenaltyDist[i]$: Valid distributions are `UNIFORM` and `NORMAL`.

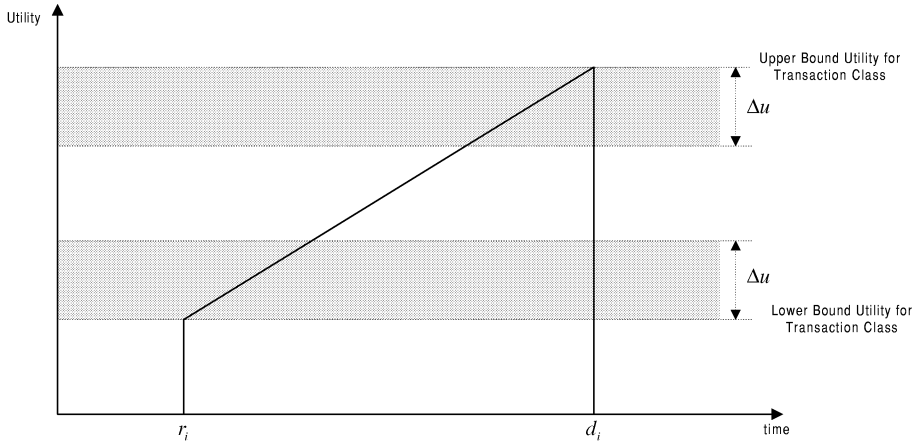


Figure C.1: An example of a linearly increasing value function.

C.1.3 Transaction Size Variables (Number of Operations)

- $MCTransMinSize[i]$: Specifies the minimum number of operations for transactions in class i.
- $MCTransMaxSize[i]$: Specifies the maximum number of operations for transactions in class i.
- $MCTransMeanSize[i]$: Specifies the mean number of operations for transactions in class i.
- $MCTransSizeDist[i]$: Specifies the distribution function. Valid distributions are `UNIFORM`, `NORMAL` and `TRIANGULAR`.

C.1.4 Arrival Frequency Variables

- $MCTransPeriodicity[i]$: Specifies the type of periodicity of transactions of class i. Valid values are `PERIODIC`, `SPORADIC` and `APERIODIC`.
- $MCTransMinPeriod[i]$: Specifies the minimum period (for sporadic transactions; it specifies the minimum inter-arrival time) for

transactions of class i.

- $MCTransMaxPeriod[i]$: Specifies the maximum period for transactions of class i.

- $MCTransPeriodDist[i]$: Specifies the distribution function for assigning period to transaction. Valid values are `UNIFORM` and `NORMAL`.

C.1.5 Soft-Deadline Transaction Variables

The following variables are only applicable to soft-deadline transactions, i.e., transactions having a secondary deadline which is later than the primary transaction deadline.

- $MCTransMinDlTol[i]$: Specifies the minimum deadline tolerance, i.e., the minimum additional time from the transaction deadline ('earliest' secondary deadline).

- $MCTransMaxDlTol[i]$: Specifies the maximum deadline tolerance, i.e., maximum additional time from the transaction deadline ('latest' secondary deadline).

- $MCTransSDlDist[i]$: Distribution function. Valid values are `UNIFORM` and `NORMAL`.

- $MCTransDecayRate[i]$: The decay rate of the penalty between the deadline and the secondary deadline of the transaction. Valid values are `LINEAR` and `EXPONENTIAL`.

C.1.6 Robustness Variables

- $MCTransRobustNoComp[i]$: Specifies number of completions of transactions of class i. This number should be relative to $MCTransRobustOutOf$.

- $MCTransRobustOutOf[i]$: Specifies the history lengh, i.e., the number of transaction executions that should be used. This number should be relative to $MCTransRobustNoComp$.

The semantics of the variables are that $MCTransRobustNoComp[i]$ out of $MCTransRobustOutOf[i]$ transaction of class i must complete. Transaction classes specified to have contingency transactions actions should specify the robustness factors to be N:N, i.e., every transaction must be completed. The current implementation does not support soft-deadline transactions to have contingency transactions.

C.1.7 Temporal Variables

- $MCTransTemporalData_B[i]$: Specifies whether the transactions of class i should have temporal constraints on data (data deadlines). Valid values are `TRUE` and `FALSE`.

C.2 Contingency Transaction Class Description

- $NumCTransClasses$: Specifies the number of contingency transaction classes.

C.2.1 Deadline Variables

The deadline of a contingency transaction is relative to the deadline of the original transaction. The following variables are defined for contingency transaction class j:

- $CTransMinDl[j]$: The minimum 'deadline tolerance' (later deadline) of the contingency transaction with respect to the deadline of the original transaction.
- $CTransMaxDl[j]$: The maximum deadline tolerance of the contingency transaction with respect to the deadline of the original transaction.

- $CTransDlDist[j]$: Specifies the distribution function used for computing the actual deadline. Valid values are `UNIFORM` and `NORMAL`.

If deadline tolerance is specified to be zero, the deadline of the contingency transaction is the same as for the original transaction. Otherwise, the deadline of the contingency transaction is distributed between d_i and $d_i + \delta_i$.

C.2.2 Value Function Variables

Invoking a contingency implies a reduced utility to the system. The underlying assumption is that the contingency transaction returns a constant value. There are two ways of representing the utility obtained by completing a transaction: specifying a 'value reduction factor', or with min- and max-values and a corresponding value-function.

- $CTransVRFactor[j]$: Expresses the value reduction factor. The actual utility obtained by executing a contingency transaction is the reduction factor times the average utility as expressed by the value function of the original transaction.
- $CTransMinVF[j]$: Expresses the minimum utility for a contingency transaction of class j.
- $CTransMaxVF[j]$: Expresses the maximum utility for a contingency transaction of class j.
- $CTransVFDist[j]$: Specifies the distribution function. Valid values are `UNIFORM` and `NORMAL`.

C.2.3 Contingency Transaction Size Variables

- $CTranMinSize[j]$: Specifies the minimum number of operations for contingency transactions in class j.
- $CTranMaxSize[j]$: Specifies the maximum number of operations for contingency transactions in class j.

- $CTranMeanSize[j]$: Specifies the mean number of operations for contingency transactions in class j.
- $CTranSizeDist[j]$: Specifies the distribution. Valid distributions are `UNIFORM, NORMAL` and `TRIANGULAR`.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering - att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.