Dynamic Transaction Scheduling and Reallocation in Overloaded Real-Time Database Systems *

J. Hansson*, S.H. Son†, J.A. Stankovic†, S.F. Andler*

* Department of Computer Science
University of Skövde, Sweden
{jorgen,sten}@ida.his.se

† Department of Computer Science
University of Virginia, USA
{son,stankovic}@cs.virginia.edu

Abstract

In real-time systems it is of paramount importance that time constraints of tasks are enforced. A tremendous amount of research has been carried out on scheduling problems associated with such systems, primarily focusing on priority assignment policies in non-overloaded systems. While static real-time systems, by definition, do not suffer from overloads, they offer limited or no flexibility and ability to adapt to new situations, often making them a poor choice for complex real-time applications. While dynamic real-time systems often meet these demands, they are prone to transient overloads.

In this paper we introduce a novel scheduling architecture with a new algorithm for dynamically resolving transient overloads, that is executed when a new transaction cannot be admitted to the system due to scarce resources. The resolver algorithm generates a cost effective overload resolution plan which, in order to admit the new transaction, finds the required time by de-allocating time among the previously admitted but not yet completed transactions. Considering the cost efficiency of executing the plan and the importance of the new transaction, a decision is made whether to execute the plan and admit the new transaction, or to reject it. the new transaction.

We consider a multi-class transaction workload consisting of hard critical and firm transactions, where critical transactions have contingency transactions that can be invoked during overloads. We present a thorough performance analysis showing to what degree the overload resolver enforces predictability and ensures the timeliness of critical transactions when handling extreme overload scenarios in real-time database systems.

* This work was performed while the first author was visiting University of Virginia, supported by the University of Skövde, Sweden and, NUTEK (The Swedish National Board for Industrial and Technical Development).

1 Introduction

In real-time systems temporal correctness is of utmost importance, normally expressed as deadlines by which a task should be finished. Deadlines which always must be met are considered to be hard critical, since missing a single deadline may have catastrophic consequences. In contrast, firm deadlines should also be met but may be missed occasionally, e.g., during transient overloads.

Some work has been carried out on scheduling of transactions having alternative actions, where the different approaches can be categorized by studying how a task is decomposed and the model for executing the decomposed tasks. In the *primary/backup model* (also referred to as the primary/alternate task model) tasks have a primary action and a backup action. The alternate is executed in order to recover a system from the failure of completing the primary action, for example timing faults of primaries [12, 15], processor failures [11], or database consistency faults [19]. Hence, in this model either the primary or the alternate completes, and the scheduler must make the decision of which one to execute. In the *imprecise computation model* [17, 18, 13] it is suggested that tasks are decomposed into one mandatory and one optional subtask, where the former computes a result that satisfies the minimum system requirement. By executing the optional task as well, the quality of the result will be increased. Hence, execution of both the mandatory part and the optional part will provide a result with no quality reduction. Mandatory tasks have hard critical deadlines and optional tasks have firm deadlines.

During transient overloads, non-critical transactions are dropped and only critical transactions are executed. Admission control filters transactions in order to ensure that admitted transactions, based on some schedulability test that considers the resource requirements of the already admitted transactions and the new transaction, are schedulable. If the new transaction cannot be admitted, the transaction is generally rejected. However, if we consider a new transaction that is critical, and therefore must be admitted, we have an obvious conflict.

Our contributions are twofold. In this paper we introduce (i) a novel scheduling architecture, that includes dynamic admission control, a transaction scheduler, an overload resolver, and a dispatcher, and (ii) a new algorithm for overload management for a multi-class transaction workload in main-memory resident database systems. Overloads are detected by the admission controller, which invokes an 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. It also determines whether it is advantageous to carry out the plan or not.

Transaction classes are differentiated by their criticality, where transactions have hard critical deadlines or firm deadlines. In addition, critical transactions have contingency transactions that can be invoked in case of overloads and executed instead of the original transaction. Our algorithm attempts to maximize the sum over all transactions of utility less any penalty imposed on the system, but more importantly, as our performance study shows, the algorithm enforces the time constraints of hard critical transactions.

2 Problem Description

2.1 Transactions and Temporal Attributes

The workload consists of a set of transactions, $\mathrm{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, where transactions have hard critical or firm time constraints, i.e., a multi-class transaction workload. In addition, critical transactions have corresponding contingency transactions, denoted $\bar{\tau}_i$, which may be invoked and executed during overloads replacing the original transaction τ_i.

The following temporal attributes related to the temporal scope constituted by a transaction τ_i are assumed to be known a priori (the corresponding temporal attributes of a contingency transaction $\bar{\tau}_i$ are indicated by a bar):

- d_i - the deadline at which the execution of transaction τ_i should be complete;
- ω_i - the worst-case-execution time of transaction τ_i (independent of the current state of the database and the system);
- κ_i - the transaction criticality of τ_i (hard critical or firm);
- b_i - the maximum amount of time that transaction τ_i can block another transaction of higher priority.

2.2 Description of the Research Problem

Current state-of-the-art scheduling algorithms featuring overload tolerance either reject transactions upon arrival (e.g., [5, 7]) or carry out only partial execution of transactions (e.g., imprecise computation tasks and incremental algorithms [17, 18, 13]). The rejection method is only applicable for real-time systems where non-critical (soft and firm) transactions are rejected at run-time, and where critical periodic transactions are statically guaranteed. The imprecise computation model can be used in critical systems, but cannot be adopted in systems where non-incremental algorithms are used, in which case tasks cannot be divided into mandatory and optional tasks (see section 1).

Generally, the contingency transaction $\bar{\tau}_i$ has significantly smaller resource and processing requirements in comparison to the original transaction τ_i. As the contingency transaction $\bar{\tau}_i$ is a substitute action, producing a result of less quality, it is of interest to minimize the number of replacements. It is therefore usually preferred to execute the original transaction τ_i if its timeliness (ability to meet its time constraints) is not jeopardized as opposed to execute the contingency transaction $\bar{\tau}_i$. In our work, the desire of completing the original transaction as opposed to the contingency transaction is represented by their value functions, describing the utility contributed to the system once the transaction completes. A value function, denoted $\upsilon_i(t)$, or $\bar{\upsilon}_i(t)$ respectively, describes the relative importance of the transaction τ_i and $\bar{\tau}_i$ in relation to other transactions in the workload. In our study, value functions are assumed to be piecewise constant. This is a simplification (for this study) of a more general value function where the utility contributed to the system may change over time.

$$\upsilon_i(t) = \begin{cases} u_i > 0 & t \leq d_i \\ -\infty & \kappa_i = \text{hard } \wedge t > d_i \\ 0 & \kappa_i = \text{firm } \wedge t > d_i \end{cases}$$

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

Hard critical transactions are sporadic or periodic, while firm transactions may be aperiodic as well as sporadic or periodic (periodic transactions are modeled as sporadic transactions with constant inter-arrival time). Upon arrival, transactions are ready to execute immediately.

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

ASSUMPTION 2 (LOAD HYPOTHESIS) The set of critical transactions is always schedulable based on the worst-case execution time of their contingency transactions.

2.3 Overload Model

In our system we have a workload consisting of a mixture of sporadic critical and aperiodic non-critical time constraints, and therefore special care is needed. Overloads potentially causing critical deadlines to be missed are the most important ones to resolve. The consequences of this is that the overload mechanism must be sensitive to the type of overload that is about to occur, i.e., what type of time constraint that is about to be missed. By performing admission control, considering the worst-case execution time ω_i of each transaction τ_i, we can ensure that the set of admitted transactions is schedulable. For uni-class transaction workloads where all transactions are either non-critical or are of equal importance, transactions are normally rejected once it is determined that they cannot be admitted given the current workload. For multi-class transaction workloads, transactions which are sporadic and critical are harder to handle. Critical transactions have to meet their deadlines, therefore they must be admitted to the system. However, if too many non-critical transactions have already been admitted, the critical transaction cannot immediately be admitted due to scarce resources, implying that resources must be de-allocated to allow the critical transaction to be admitted. We call this process overload resolution, and this work is carried out by the overload resolver. In this work two types of actions are used in combination to resolve overloads:

1. Controlled dropping of non-critical transactions. This strategy is carried out by the overload resolver both at the admission control level, i.e., rejecting newly arrived transactions, and at the scheduler level, i.e, preempting and aborting currently running transactions.

2. Invocation of contingency transactions for critical transactions, replacing the original transactions. This strategy is carried out by the overload resolver both at the admission control level and the scheduler level.

Controlled dropping is being used to get overall better utility from the system, versus if all non-critical transactions are being dropped in which case we get safety but poor utility.

Our real-time database system model incorporates two processing elements, one dedicated for performing scheduling activities (admission control, overload management and scheduling of real-time transactions) and one for executing transactions [1, 14]. There are several reasons that warrant the use of a dedicated processing element for scheduling and overload management. First, scheduling processors offload both the scheduling algorithm and other operating system overheads from the application tasks, both for speed and so that external interrupts and operating system overhead do not cause uncertainty in the execution of transactions and tasks. 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 [4]. 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 [16, 4].

The database is main-memory resident, and hence, the blocking of transactions due to disk delays are avoided. We use a real-time optimistic concurrency control [9, 10].

3 Algorithm Description

3.1 Scheduler Architecture

The architecture consists of a dynamic admission controller, a transaction scheduler, an overload resolver, and a dispatcher. The admission controller tests for schedulability of new transactions upon their arrival. It acts as a transaction filter guaranteeing that the transaction scheduler will always be able to find a feasible schedule.

In case a new transaction cannot be admitted immediately by the admission controller, the overload resolver is invoked in order to examine whether reallocation of resources should be performed. The transaction scheduler uses Earliest Deadline First (EDF) and Stack Resource Policy (SRP [2] described below) for scheduling of the admitted transactions and preventing chained blocking. The overload resolver is invoked when an original transaction τ_i cannot be admitted to the system, in which case the resolver initializes a negotiation of the requirements of the previously admitted transactions and the new transaction. The overload resolver develops an *overload resolution plan (ORP)*, evaluates its cost efficiency and executes the plan if it is considered to be worthwhile. The dispatcher performs dispatching according to the schedule determined by the transaction scheduler.

The admission controller performs admission based on the worst-case execution times of new transactions. Once transactions are admitted, the required amount of processing time is reserved.

We have adopted the Stack Resource Policy (SRP) [2], as an improvement to priority ceiling protocols (PCP), in order to prevent blocking and multiple priority inversion. SRP relies on two premises: to prevent deadlock, a job should not be permitted to start until the resources currently available are sufficient to meet the maximum requirements of the job, and to prevent multiple priority inversion, a job should not be permitted to start until the resources currently available are sufficient to meet the maximum requirement of any single job that might preempt it. In SRP ceilings are static and based on preemption levels as opposed to PCP where ceilings are dynamic and based on the scheduling priority. Since SRP ceilings are static, SRP can be applied directly to EDF scheduling without resort to dynamic recomputations of ceilings.

With SRP, the blocking time b_k is given by the maximum worst-case execution time of the longest nontrivial critical section of every transaction τ_k such that $d_i < d_k$. This maximum includes the worst-case execution time of all the critical sections of other transactions that might subject τ_i to priority inversion. The following condition should be satisfied in order to guarantee the schedulability of a set of periodic and aperiodic transactions using SRP together with EDF [2]:

THEOREM 1 (BAKER, 1991) *A set of n (periodic and aperiodic) processes is schedulable by EDF scheduling with SRP semaphore locking if*

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

Proof. See [2][p.83].

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

3.2 Overload Resolver

The basic idea behind this is to generate a plan that resolves the impending overload by de-allocating time from previously admitted transactions. In the case of an overload, the resource reservations are scrutinized, and they may be reduced by substitution or de-allocated by dropping. Hence, transactions that are admitted are given a prognosis that they will get their desired level of resources. In the worst scenario, given the load hypothesis, this implies that non-critical transactions are terminated and that critical transactions may be replaced by their contingency transactions (see section 2.3). We call these *overload resolutions actions (ORAs)*. Dropping and replacing transactions result in a utility loss relative to the initial expectations, and possibly, wasted processing time if the transactions have already started to execute. Hence, it is of interest to minimize the utility loss. Note, a new transaction will eventually (when complete) give some utility to the system, and this must be contrasted with the utility loss caused due to de-allocating enough resources in order to admit the new transaction.

If we are about to resolve an impending overload, it is important to have an understanding of in which time interval the overload is occurring. As we will see, the reason for this is that in order to resolve an overload, not necessarily any transaction can be dropped and thereby make the workload schedulable. Consider a set of sporadic and/or aperiodic transactions arriving in a system which uses EDF. EDF is known not to handle overloads very well, causing a domino-effect of missed deadlines. Let us define two intervals, namely, a total overload interval and a critical overload interval.

Let $\mathrm{T}_{\mathcal{A}}$ denote the set of admitted transactions, and let τ_η represent a new transaction coming in to the system. Consider the situation when the set of transactions $\mathrm{T}_{\mathcal{A}}$ can be feasibly scheduled with EDF, but where $\mathrm{T} = \mathrm{T}_{\mathcal{A}} \cup \{\tau_\eta\}$ is not schedulable. The *total overload interval* $[t_1, t_2]$ denotes the total time the overload will last, i.e., from the current time (t_1) until t_2 which is the latest deadline of those transactions missing their deadline. Admitting transaction τ_η will delay the execution for ω_η time units (approximately) of those transactions having lower priority than τ_η, and in the worst case they will miss their deadlines.

In our approach, de-allocation of processing time is done in an interval referred to as the critical overload interval. Informally, the *critical overload interval* starts once the overload is detected at time t_1 and ends at the time of the deadline of the first transaction having the smallest slack (before τ_η is admitted) among those transactions having a priority lower than τ_η. Note, the critical overload interval is smaller or equal to the total overload interval, and resolving the overload here is guaranteed to resolve it in the total overload interval. However, de-allocating time outside the critical overload interval will not resolve the overload.

More precisely, the resolver (i) determines the critical overload interval and computes the amount of processing time that needs to be de-allocated in the interval in order to resolve the overload; (ii) generates an overload resolution plan (ORP) which consists of a set of ORAs that de-allocate resources from admitted transactions; and (iii) decides whether it is advantageous to carry out the ORP, considering the relative utility loss/gain of executing the ORP and accepting the new transaction in comparison to simply rejecting it (firm) or substituting (hard critical).

From the set of eligible ORAs, 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 by definition not be considered acceptable. Hence, any ORP containing ORAs with infinite penalty will not be performed.

3.2.1 Computing 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 x on an arbitrary transaction τ_i, is computed as follows. When dropping a transaction τ_i, the amount of time de-allocated is:

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

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

$$\psi = \begin{cases} b_i - max\{b_j | i \neq j\}; & \text{if } b_i > max\{b_j | i \neq j\} \\ 0 & \text{otherwise} \end{cases}$$

Since we are using SRP we know that a high-priority transaction may be blocked only once by low-priority transaction. b_i is thus the worst-case execution time for the longest critical section among the low priority transactions. If transaction τ_i is the transaction with the longest critical section, then b_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 remaining critical section of τ_i (b_i) and the second longest remaining critical section of transaction τ_j (b_j).

The time de-allocated when an original transaction τ_i is replaced by its contingency transaction $\bar{\tau}_i$ can be computed in a similar way. Similarly, 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:

$$\xi_i^r = \zeta_i - o_i - \bar{\omega}_i + \psi$$

where

$$\psi = \begin{cases} b_i - max\{\bar{b}_i, b_j | i \neq j\}; & \text{if } \bar{b}_i < b_i \text{ and} \\ & b_i > max\{b_j | i \neq j\} \\ 0 & \text{otherwise} \end{cases}$$

In case the ORP is composed of several ORAs, the amount of time saved due to non-occurring blocking depends on which transactions that are selected to be dropped or replaced, and hence, cannot be computed before ORAs are selected. The following example shows why. Consider the following set of transactions $\{\tau_i, 1 \leq i \leq n | d_{i-1} < d_i\}$, further assume that τ_k has the longest critical section, i.e., $b_k > max(b_i | i < k)$. This implies that τ_k may block another transaction for b_k time units. Hence, if we decide to drop τ_k we will save $b_k - b_j$ time units, where b_j represent 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 b_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 b_k time units. In conclusion, at the time that ORAs are generated there is no knowledge of which ORAs will be selected later on to be part of an ORP, which makes it hard to estimate any 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, i.e., ψ is initially considered to be negligible. The computation is determined at the time when ORAs are generated. Computing the time saved due to blocking that no longer will occur is postponed until the 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 then know which transactions that are subject to be dropped or replaced and how these transactions relate to other transactions.

3.2.2 Computing Utility Loss Caused by an ORA

The amount of *utility loss*, denoted γ_i^x, as a result of a specific ORA x on an arbitrary transaction τ_i, is computed as follows. When dropping a transaction τ_i, the utility loss is given by the penalty of missing the deadline d_i, i.e.:

$$\gamma_i^d = \upsilon_i(t \leq d_i) - \upsilon_i(t > d_i)$$

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

$$\gamma_i^r = \upsilon_i(t \leq d_i) - \bar{\upsilon}_i(t \leq \bar{d}_i)$$

3.2.3 Algorithm for Generating the Overload Resolution Plan

To select a subset of eligible ORAs, such that utility loss is minimized, is an NP-hard optimization problem. Our heuristic algorithm selects ORAs by relating the utility loss caused by a certain action to the amount of resources that is de-allocated by it, i.e., we compute the *utility loss density* (γ_i^x / ξ_i^x) for each ORA, and, let the set of actions be ordered by their utility loss density.

The overload resolution algorithm determines the best set of ORAs to select in order to save a minimum of execution time in the critical overload interval. The algorithm uses the utility loss density for selecting the actions, and then computes the total utility loss. The algorithm consists of the following steps:

1. Generate the set of possible ORAs actions and compute their utility loss density.

2. Iterate through the ORAs in order of decreasing utility loss density, and add them to the ORP as long as the total amount of saved time by the new ORP (overload resolution plan) does not exceed the time required.

3. From the remaining actions, add the action with minimum utility loss, such that the required time is met or exceeded, to the set of selected actions.

4. Remove any now unnecessary actions from the set of selected actions. Start with the ORA with the highest utility loss, such that the required time is still met or exceeded.

By selecting an ORA by its utility loss density, we ensure that the de-allocated time is cost efficient with respect to other ORAs. Given 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 execution time of the contingency, replacing the critical transaction by its contingency transaction is not justified in the general case[1].

Note that in the case 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 if the original transaction would have been admitted, but it also contributes with less utility. Hence, we generate up to three ORPs and compute the relative cost of implementing them: (i) admitting the original transaction; (ii) admitting the contingency transaction; and (iii) rejecting the transaction (non-critical transactions only). The most cost effective approach will then be selected, i.e., the one with the best overall utility gain/loss. Let $\Gamma(x)$ denote a function returning the amount of utility lost due to de-allocating x time units. The change in utility, denoted Φ, can be computed as follows, and the ORP with the lowest utility loss (or highest utility gain) will then be selected. Let s denote the smallest slack among the transactions active in the critical overload interval and that have a deadline later than transaction τ_η.

$$\begin{array}{ll} \text{admit } \tau_\eta: & \Phi_1 = \upsilon_\eta(t \leq d_\eta) - \Gamma(\omega_\eta - s) \\ \text{admit } \bar{\tau}_\eta: & \Phi_2 = \bar{\upsilon}_\eta(t \leq \bar{d}_\eta) - \Gamma(\bar{\omega}_\eta - s) \\ \text{reject } \tau_\eta: & \Phi_3 = \upsilon_\eta(t \leq d_\eta) - \upsilon_\eta(t > d_\eta) \end{array}$$

[1] 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, then replacing the original transaction could be justified.

4 Performance Analysis

In order to evaluate the algorithm, we have developed a simulator, modeling the performance of the overload resolution algorithm in a centralized real-time database system.

Our simulations include the results from two comparative 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 changes policy depending on the severity of the overload. Once an overload occurs, non-critical transactions are unconditionally rejected by the admission controller, leaving only critical transactions to be admitted. If 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 mode switches: (i) reject all non-critical transactions; and (ii) admit only contingency transactions. It should be noted that the sole purpose of BL is to serve as a comparison baseline for our experiments, that is, indicating when no feasible schedule can no longer be found even though only contingency transactions are admitted and scheduled. In fact, BL is inappropriate as an algorithm since it performs the first mode switch when it has failed to admit a non-critical transaction. The second mode switch is triggered when an original and critical transaction cannot be admitted.

4.1 Simulation and Workload Parameters

We have evaluated the performance of the overload resolution algorithm by modeling certain database system parameters and a number of workloads, using values for simulation parameters that represent what we believe is realistic, as shown in tables 1 and 2. As noted in section 2.3, we use dual processors, where one processor is dedicated for scheduling services and the other is dedicated for transaction processing.

Table 1. Database system parameters

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

We have defined the workload to consist of two transaction classes (hard critical and firm transactions), where each class is 50% of the total workload. The critical class of transactions has contingency transactions that differ in size and utility from the original transaction. Beyond that, the two transaction classes are equal. Each transaction class is defined as follows ('$\leftarrow$' means that the corresponding value is the same as for the original transaction τ_i):

Table 2. Transaction workload parameters.

Transaction Class	Hard critical original	Hard critical contingency	Firm 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

Aperiodic transactions arrive according to a Poisson distribution. The 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 access a number of pages that are selected uniformly within the database.

4.2 Results

Each experiment was conducted by running a series of three simulations and summarizing them. We use completion ratio (CR) as our primary metric, i.e., the ratio of the number of transactions that successfully complete and the total number of transactions requesting resources. Each data point is the average completion ratio with 95% confidence intervals (values are shown in figure 1).

We have studied the completion ratio of the three algorithms (overload resolution algorithm - denoted OR in the graphs, EDF, and BL) as shown in figure 1. We can see that 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. EDF's inability to handle overloads, causing a domino-effect of missed deadlines, is well-known and is shown in figure 1. Since both classes share the same characteristics, no transaction class is favored, thus, the decay rates of the completion ratio are similar.

Since it is of paramount importance that the time constraints of the critical transactions are met, we observe that the overload resolution algorithm satisfactorily enforces this requirement. During extreme overloads it performs close to optimal (in terms of meeting time constraints for critical transactions) as set out by the baseline BL. The algorithm ensures the timeliness of critical transactions by gradually increasing (from zero) the numbers of critical transactions being replaced by their contingency transactions (see figure 1c) and gracefully dropping/rejecting firm transactions, which is a significant improvement compared to both the baseline and the EDF algorithm.

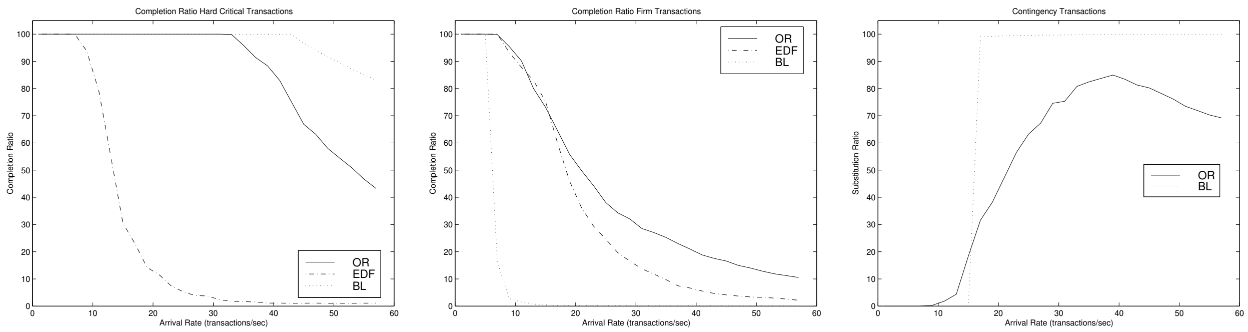
The reason why the algorithm does not achieve optimal results during extreme overloads is the sporadic nature of the workload and the fact that the scheduler is not clairvoyant. Original transactions are admitted when possible, since they contribute a high utility to the system. The drawback of this is that when additional transactions arrive, a feasible schedule may no longer be found. Although the original transactions are replaced by their contingency transactions, unnecessary processing time has been lost. This can be overcome to some degree by having the workload monitored, and when the workload goes beyond a certain level the admission controller is notified to only admit contingency transactions.

In order to verify the robustness of overload resolution algorithm, additional simulations have been performed where different parameters have been varied, e.g., slack factor, importance value, and number of database operations, to simulate different workload scenarios. Due to the limited space in this article, we only include the simulation results from the experiment where the relative utility of the contingency transaction in comparison to the utility of the original transaction was varied. In this experiment we have used the same workload as specified in table 2, except that the utility obtained by the contingency was varied, where $\bar{v}_i(t) = c * v_i(t)$ and c is varied between 0.0 (no utility is obtained for completing the contingency transaction $\bar{\tau}_i$) and 1.0 (there is no reduction in utility when completing the contingency transaction as opposed to the original transaction). The results can be seen in figure 2. We observe that the completion ratio of hard critical transactions is intact for $c > 0.25$ (for arrival rates within the operational envelope).

5 Related Work

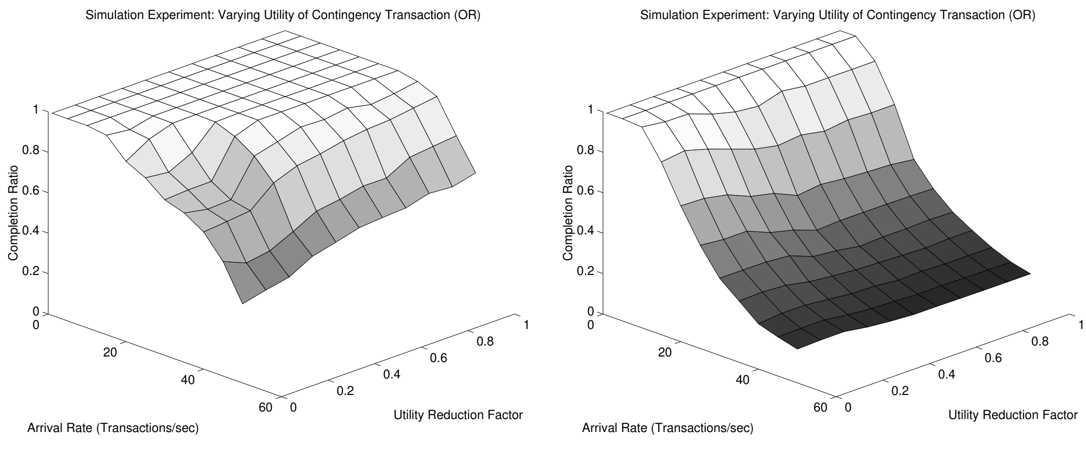
Chetto and Chetto [6] have suggested that alternative actions are scheduled according to Earliest Deadline Latest (EDL), implying that alternative actions are executed as late as possible. This gives the primary actions the maximum amount of time possible to execute. The time reserved for the alternative action is released once the primary action completes, increasing the processor availability. In contrast to our model, they assume that the worst-case execution time is not known for the primary action (worst-case execution time is known for alternative actions). Moreover, they do not consider sporadic and aperiodic tasks, but only periodic tasks where all tasks have the same criticality. Liestman and Campbell [12] have focused on how to execute the maximum number of primary periodic tasks on a uni-

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



(a) Hard critical transactions (b) Firm transactions (c) Contingency transactions

Figure 1. Completion ratio of successfully completed transactions in relation to the number of transactions requesting resources. Graph (a) and (b) shows the completion ratio of hard critical and firm transactions. Graph (c) shows the percentage of hard critical transactions that have been replaced. Note, the arrival rate as presented in the graphs denotes the total number of transactions arriving to the system per second



(a) Hard critical transactions (b) Firm transactions

Figure 2. Completion ratio of successfully completed transactions in relation to the total number of transactions requesting resources.

processor system. However, they assume prior knowledge of the worst-case execution time of both primary and alternative tasks.

Nagy [15] considers the deadline scheduling problem where compensating transactions, with known worst-case-execution times, are used as safe mechanism for bailing out if the primary transactions are not able to finish within their deadlines due to unknown processing requirements. Admitted transactions are guaranteed to complete either by successful commitment of the primary transaction or by safe termination of the compensating transaction, i.e., compensating transactions are not used for resolving overloads. Instead, overloads are resolved at the admission control level by rejecting new transactions at submission time. In contrast to our work, alternative actions are used as one mechanism for resolving overloads by either saving time or buying extra time (the other mechanism is rejection of transactions). However, in our system overloads are resolved by the overload resolver both at the admission control level and the scheduler level. In our work, the deadline of a contingency transaction may also be later than the deadline of the original transaction.

Liu *et al.* [13] defined 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 contrast to our approach, the original transaction and the alternative action has a 0/1 constraint, either the original transaction is executed or the alternative. Hence, during overloads the alternative actions is executed as opposed to Liu *et al.* 's approach where only the mandatory tasks are executed.

Buttazzo *et al.* [5] investigated the performance during overloads of the Earliest Deadline First, Highest Value First, Value Density and the *Mix* dynamic scheduling algorithms. Two derivatives of each algorithm were developed to enforce the notion of guarantee and robustness, where the class of guarantee algorithms uses an acceptance test invoked at admission of new tasks. The robust algorithms perform a guarantee test upon task activation, a rejection based on the importance value of the tasks, and combined with a resource reclaiming mechanisms taking advantage of early task terminations [5]. Tasks have a weight value reflecting the importance level of the individual task. Their work suggests that scheduling by deadline and rejecting by value is the most effective strategy for a wide range of overload conditions.

In contrast to Buttazzo *et al.* [5], Haritsa *et al.* [8], and Baruah *et al.* [3], our work addresses additional complexities with respect to the multi-class nature of the workload (hard critical and firm deadlines), and the additional transaction element, namely the contingency transactions, requiring more sophisticated admission control and rejection algorithms.

6 Discussion and Future Work

While hard critical and firm transactions offer no deadline tolerance, soft deadlines are considered less stringent, allowing some degree of tardiness. We will now discuss how soft transactions with deadline tolerance can be incorporated into our model and how tardiness is controlled during overloads. Our proposed algorithm computes the amount of time to de-allocate and which transactions to replace or drop among those transactions being active in the critical overload interval. The notion that soft deadlines should meet their deadlines but that it is acceptable if they are occasionally missed, suggests that soft transactions should be scheduled based on their deadline, but that tardiness should be controlled during transient overloads.

If a soft transaction has its deadline inside the critical overload interval, but where the secondary deadline of the soft transaction is outside the critical overload interval, then parts of the transaction can be postponed to take place past its deadline but also outside the critical overload interval. This may save time inside the critical overload interval. Hence, we have a third type of ORA, namely, postponing the execution of a transaction. Due to limited space we have excluded formulas for computing the amount of time saved if transactions are postponed. We intend to adress this in future papers.

In contrast to the imprecise computation model (see section 1), we have focused on how and when to *replace* critical transactions, as opposed to only *partially execute* critical transactions. We will now briefly discuss how the imprecise computation model can be incorporated in our scheduling model. By defining mandatory transactions to be hard critical transactions without contingency transactions, and further, define optional transactions to be non-critical transactions with firm deadlines, imprecise transactions can be admitted and scheduled by the overload resolver. However, the imprecise computation model has a precedence requirement between mandatory and optional parts, since the optional part should not execute before the corresponding mandatory part has completed. While our algorithm does not explicitly support the representation of precedence constraints, this casual chain can be modeled by letting mandatory transactions, once they complete, releasing their optional transactions.

7 Conclusions

In this paper we have presented a novel overload resolution strategy based on a scheduler architecture and with a dynamic overload resolution algorithm for handling transient overloads in real-time database systems. The architecture consists of an admission controller, a transaction scheduler, the overload resolver and a dispatcher. Our algorithm is designed for handling multi-class transactions, i.e., transactions of different criticality, where critical transactions have a contingency transaction.

Transactions are scheduled using EDF, blocking is handled by Stack Resource Policy (SRP) [2]. Admission considers the schedulability of a transaction given its worst-case execution time and the potential time it may be blocked. If a transaction cannot be admitted with its original resource requirements, the overload resolver is invoked. At a high level, the overload resolver computes the required amount of time that needs to be de-allocated, develops a plan that de-allocates the needed amount of time, and carries out the plan if it is determined to be advantageous.

We have implemented the algorithm and conducted a simulation-based performance analysis. The results show that the algorithm (i) gracefully degrades the performance during overloads by increasing the number of contingency transactions replacing the original transaction, and dropping non-critical transactions controllably; (ii) ensures the timeliness of critical transactions (below a certain operational envelope); and (iii) produces near-optimal results.

References

[1] S. 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.

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

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

[4] 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*, to appear in 1998.

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

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

[7] 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.

[8] 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.

[9] 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*, 1990.

[10] 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.

[11] C. 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.

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

[13] 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 A. van Tilborg and G. M. Koob, editors, *Foundations of Real-Time Computing - Scheduling and Resource Management*, chapter 8. Kluwer Academic Publishers, 1991.

[14] J. Mellin, J. Hansson, and S. F. Andler. Refining timing constraints of applications in DeeDS. In 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, chapter 18, pages 325–343. Kluwer Academics, 1997.

[15] S. C. Nagy. *Admission Control and Scheduling Strategies for Real-Time Database Systems*. PhD thesis, Department of Computer Science, Boston University, 1997.

[16] 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.

[17] W.-K. Shih, J. W. 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.

[18] 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.

[19] 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.