Providing Real-Time Response, State Recency and Temporal Consistency in Databases for Rapidly Changing Environments

Anindya Datta, Igor R. Viguier

May 8, 1997

TR-12

A TimeCenter Technical Report

Title	Providing Real-Time Response, State Recency and Temporal Consistency in Databases for Rapidly Changing Environments Copyright © 1997 Anindya Datta, Igor R. Viguier. All rights reserved.
Author(s)	Anindya Datta, Igor R. Viguier
Publication History	To Appear in Information Systems A TIMECENTER Technical Report

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta
Sudha Ram

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, Polytechnic University, New York, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

Any software made available via TIMECENTER *is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.*

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors, The Rune alphabet (second phase) has 16 letters. They all have angular shapes and lack horizontal lines because the primary storage medium was wood. However, runes may also be found on jewelry, tools, and weapons. Runes were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

Databases have been proposed as the chosen platform to develop automated control systems for environments where the state of the system changes rapidly, and actions need to be taken with timing constraints, such as stock trading, air traffic control, network management and process control to name a few. The workload in such systems consist primarily of two types: (a) **updates**, which report the state of the environment to the database and arrive very frequently, and (b) **transactions** which arrive with deadlines. In this paper we develop algorithms to install these updates rapidly enough to ensure "state recency" while allocating sufficient resources to transactions to minimize tardiness. We also require that these transactions read temporally consistent data. In summary, this paper contributes in the following three interrelated issues:

1. First and foremost, we look at some interesting problems regarding amalgamating temporal and real-time databases
2. We look at simultaneous satisfaction of state recency and temporal consistency which are central issues in the class of systems considered in this paper, i.e., rapidly changing systems
3. We design algorithms that attempt to achieve equilibrium with regard to the conjoint processing of frequently arriving state updates and time constrained transactions

1 Introduction

Consider an environment where the state of the system changes rapidly, and actions need to be taken with timing constraints. Examples of such systems, termed *rapidly changing systems*, are abundant – stock trading, air traffic control, network management and process control to name a few. Databases have been mandated as the chosen platform to develop automated control systems for such environments due to their data intensive as well as "data driven" nature [12]. These databases serve as the "window" to the environment for control applications and thus must reflect the true state of the external environment as closely as possible. This is a non-trivial task by definition, as the state of these environments change rapidly, and consequently, state updates arrive frequently. While processing these updates, the system must also, simultaneously, ensure time cognizant processing of transactions. This complicates the problem of allocating system resources to both state updates and transactions. One must strive to strike a delicate balance in this allocation process – processing updates at the expense of transactions will mean very few transactions will finish on time, while sacrificing updates to minimize tardy transactions will mean the database will not be representative of the state of the outside environment.

Transaction processing, in addition to being real-time in nature, is further complicated by the need to read *snapshot consistent* or *temporally consistent* data. As an example consider a network management database that serves as the basis of applications managing and controlling a telecommunications network [23, 4]. Such a database needs to process a large number of state updates – a commercial telecommunications service provider handles around 30GB of data per day [9]. The database also needs to provide real time transaction support, e.g., a sample transaction may be: `update routing profile in all routers in location XYZ within 60 seconds`. This may involve the running of procedure `update_routing_profile` that requires the reading of a number of attributes. The system must ensure that the values provided must have co-occurred, i.e., come from the same *snapshot*. Otherwise the resultant action may not have the intended effect. Also, to ensure the best possible effect of transactions, the system is often required to make available the *most recent* values of state variables. For example, consider the following transaction; `read(Microsoft stock price), read(IBM stock price), make trading decision`. Clearly the trading decision depends on the relationship between Microsoft and IBM stocks. The quality of such a decision would be enhanced if it were based on recent stock prices rather than old stock prices.

Based on the problem identified above, we consider in this paper strategies to provide the following capabilities in databases used to model and control the aforementioned environments.

1. The database must provide as recent a picture of the outside world as possible. We call this *state recency*. In addition to providing state recency, the database must also provide other, previous states. This is essential as transactions may ask for values of data items at current as well as previous moments in time. Thus, not only is it necessary to maintain *current views* of the system, historical views are important as well.

2. The database must ensure transactions read temporally consistent data.

3. The database must ensure real-time transaction processing, i.e., transactions should be completed by their deadlines.

The three required capabilities enumerated above show that these databases must amalgamate *real-time* as well as *temporal* characteristics. Thus, this paper may be considered as a study of transaction processing in *real-time, temporal databases.*

The rest of the paper is organized as follows. In Section 2 we provide a brief review of relevant prior work and summarize the contributions of this paper. In Section 3 we describe the model of data and workload of the class of database systems we consider in this paper. In Section 4 we describe our strategies to enforce recency and temporal consistency, while Section 5 explores scheduling algorithms. Subsequently, we describe our simulation model in Section 6, discuss performance of the proposed system in Section 7 and conclude in Section 8.

2 Related Work and Contribution of this Paper

While there has been much work done on *Temporal Databases* as well as *Real-Time Database Systems* (RTDBSs) in isolation, very little has been reported on their synthesis.

Initiated by the pioneering work in [7, 48], *temporal databases* have been the focus of substantial work by researchers attempting to enrich conventional database semantics with that of time [5, 10, 11, 18, 19, 20, 28, 43, 44, 46, 49]. A large majority of these papers have explored either extensions to conventional data models such as relational or object-oriented or have concerned themselves with the enrichment of conventional languages such as relational algebra and SQL with temporal augmentations. Note that none of this work has considered a real-time context along with the temporal dimension. Also, while notions of *temporal consistency* have been recognized [45], not much is reported on efficient means of enforcing such consistency requirements with regard to transactions. In general there is not a whole lot reported on transaction processing in temporal databases. A few notable exceptions are [34, 42], which are nice papers on temporal query processing, and [15] which is one of the very few papers that we are aware of, on temporal transaction processing.

RTDBSs have seen substantial research effort as well in recent years. Much of this effort has been focussed towards developing high performance scheduling algorithms [1, 2, 14, 21, 25, 31, 37] as well as concurrency control algorithms [6, 22, 24, 26, 27, 33]. Typically, performance has been characterized as the ability to reduce transaction tardiness. None of this work has been performed with temporal consistency in mind.

Even though not much is reported on the confluence of temporal and RTDBSs, the need for such an amalgamation has been recognized by researchers (e.g., see [8, 39]). One very recent nice work that deals with some issues related to those discussed in this paper is [3]. However, this paper considers a different environment than we do. For example, this paper considers systems where only *current views* (as opposed to *historical views*) are needed. Consequently, only one version (the most recent one) need to be stored for each data item. Thus, temporal consistency is not an issue in [3]. We, on the other hand, consider the satisfaction of temporal consistency by supporting the existence of historical views (i.e., many versions need to be stored). Furthermore, [3] assumes that exact execution times of transactions are known. We do not make any such assumption. Additionally, [3] considers a memory resident database while we consider a disk resident one, which gives rise to certain new issues as discussed later in this paper.

The final vindication of this work comes from some very recent work that identifies the need to incorporate both capabilities, i.e., temporal and real-time [13, 40]. Note again that these papers are mostly high level papers, talking broadly about the need for synthesis, rather than providing detailed algorithms to achieve such synthesis.

In summary, this paper contributes in the following three interrelated issues:

1. First and foremost, we look at some interesting problems regarding amalgamating temporal and real-time databases

2. We look at simultaneous satisfaction of *state recency* and *temporal consistency* which are central issues in the class of systems considered in this paper, i.e., *rapidly changing systems*

3. We design algorithms that attempt to achieve equilibrium with regard to the conjoint processing of frequently arriving state updates and time constrained transactions

3 Data and Workload Model

In this section we describe the data and workload characteristics in databases that model rapidly changing environments.

3.1 Data Model

We recognize two primary classes of data: *sensor data* and *non-sensor data.*

Sensor Data are those which are reported by sensors distributed throughout the environment being modeled, e.g., a telecommunications network. Sensor data, collectively, describe the state of the system. Examples of sensor data include link utilization and node queue lengths from the network management domain, stock and commodity prices from the financial trading domain and temperatures and pressures from a chemical process control domain. Typically system state changes very rapidly, leading to the generation of a large amount of sensor data. For instance, a typical network control center receives more that 30 GB of sensor data per day [9].

Non-Sensor Data are control and structural parameters of the system, e.g., split ratios in communication switches and network topology in network management, buying and selling volumes in financial trading, and degree of valve closures in process control. Such information is updated on explicit instruction to the system (by the user or automatically) and changes much less frequently than sensor data. Typically, non-sensor data are updated in order to control unacceptable behavior of the system by monitoring sensor data values.

Aside from the semantic classification of data, our data model assumes data is versioned. We shall use the notation D_i to refer to data item i and employ the notation D_i^v to denote version v of data item D_i. Sensor data items are updated frequently, signifying that new versions of sensor data items will be created often. New versions of non-sensor data items would be created far less frequently than sensor data items. The frequency of version creation makes sensor data items *rapidly changing* and consequently much harder to deal with than non-sensor data items. In fact, conventional methods are quite suitable for handling non-sensor data processing. In this paper, our primary concern is the handling of sensor data.

3.2 Workload Model

Two distinct workload types can be identified in the class of database systems under consideration: (a) transactions that update sensor data, termed *updates* [3], and (b) transactions that update non-sensor data, having read both sensor and non-sensor data, termed *transactions.*

Updates are sent to the system by sensors and arrive very frequently, reflecting the rapidly changing nature of the outside environment. Two types of updates have primarily been discussed in the literature: *periodic* and *aperiodic*. Periodic updates involve sensors reporting values of sensor data items at regular intervals, regardless of whether these values have changed or not. Aperiodic updates usually involve sensors reporting values only when they have changed – these updates arrive with unpredictable frequencies. Periodic updates are somewhat easier to deal with because of their predictable frequency. In this paper however, we consider aperiodic updates, which represent a harder and more general case. Thus, our model of update is as follows: *an update is a singleton write of the form* `write(`D_i`)` *that is sent by a sensor to report the change in the value of data item* D_i. To differentiate updates from transactions we denote an update on a data item D_i by $U(D_i)$. Note that updates exclusively write sensor data. In the class of systems that we consider, updates arrive with very high frequency. If an update executes successfully (i.e., the corresponding data item is written in the database), we say the update is *installed*, following the terminology of [3]. Installation of $U(D_i)$ results in the creation of a new version of D_i.

Transactions are modeled in the usual fashion, i.e., a partial ordering of read and write operations. There are restrictions on the read and write sets of transactions. A transaction may read both sensor as well as non-sensor data, but may only update non-sensor data. Essentially transactions model *control actions*, i.e., actions undertaken in order to modify control parameters (which are non-sensor data items) in the system. Figure 1 below shows the access sets of transactions and updates.

Sensor Data	**Non-Sensor Data**
Updates + Transaction Reads	Transaction Reads + Transaction Writes

Figure 1: Access Sets of Transactions and Updates

A unique feature of transactions in databases for rapidly changing systems is that instead of simply requesting values of data items, such requests also need to retrieve data values that were valid at particular instants in time. For example, a transaction may be of the following form: `read(`D_i`), read(`D_j`) at time = 100`. The reader can easily understand the need for such timestamp attachments to transactions by considering an example from the network management domain: if an alarm was logged at time t_1, and subsequently a transaction was submitted to respond to that alarm, the transaction would very likely wish to view values at t_1. Thus in our model, transactions have two associated timestamps, a *transaction timestamp* and a *request timestamp*. Similarly, updates are associated with *update timestamps* and data item versions are associated with *data timestamps.*

Definition 1 *The* **transaction timestamp** *of T_i, denoted by $TTS(T_i)$, denotes the time at which T_i was generated.*

Definition 2 *The* **request timestamp** *of T_i, denoted by $RQTS(T_i)$, denotes the time at which T_i requires the values of the items in its read set.*

Definition 3 *The* **update timestamp** *of $U(D_i)$, denoted by $UTS(U(D_i))$, denotes the time at which $U(D_i)$ was generated.*

Definition 4 *The* **data timestamp** *of version v of data item D_i, i.e., D_i^v, denoted by $DTS(D_i^v)$, denotes the time at which the state of D_i in the outside environment corresponded to the value of D_i^v*

In our aperiodic update model, whenever the value of D_i changes in the real world, an update $U(D_i)$ is generated. When this update is installed and creates a new data version the timestamp of the new version is set equal to the timestamp of the update, i.e., if $U(D_i)$ created D_i^v, then, $DTS(D_i^v) \leftarrow UTS(U(D_i))$.

We recognize that transactions may want to read the most recent values or older values. Therefore, the request timestamp may assume any value subject to the condition $RQTS(T_i) \leq TTS(T_i)$. Also note that an update will typically arrive at the system at a time later than its timestamp. This phenomenon has been termed *retroactive update* [30]. This reflects the fact that an update ages between the time it is generated by a sensor and the time when it is received by the system. Such aging is often the result of network delays.

Three requirements must be satisfied with regard to the successful execution of transactions:

1. **Recency:** The values read by a transaction must be *recent* with respect to its request timestamp. This is required as the quality of the control decision the transaction will implement is directly related to the age of the data items it reads.

2. **Temporal Consistency:** The values read by the transaction must be temporally consistent, i.e., they must have co-occurred.

3. **Timely Processing:** Transactions arrive with deadlines and are assumed to have no value if not processed within that deadline.

In the subsequent sections we go into details of satisfying these requirements. Table 1 below summarizes the notation used in this paper. Note that some of the notation given in Table 1 appears later in the paper.

4 Recency and Temporal Consistency

As previously mentioned, the basic motivation in enforcing recency and temporal consistency is that the system must report values close to the time at which such values are requested, and when some decision needs several values as input, such values must be temporally consistent, i.e., they must have co-occurred in the system. Ideally, we would like to provide *perfect recency* (i.e., if the

Notation	Description
D_i	Data item i
D_i^v	Version v of data item i
$U(D_i)$	Update transaction wishing to update D_i
TTS(T)	Transaction timestamp of T, indicating when the transaction was generated
RS(T)	Read set of T
WS(T)	Write set of T
RQTS(T)	Request timestamp of T, i.e., the time at which T wants to retrieve the values of elements in *RS(T)*
UTS(U(D_i))	Update timestamp of update $U(D_i)$, indicating when $U(D_i)$ was generated
DTS(D_i^v)	Timestamp of D_i^v. If $U(D_i)$ created D_i^v, then, *DTS(D_i^v)* $\leftarrow$ *TS(U(D_i))*
MTD	Maximum tolerable delay
RI	Recency interval
ATI	Acceptable timestamp interval

Table 1: Notations used in this paper

value of D_i is desired at time t_i, we would like to provide the version that represents the value which occurred precisely at t_i) and *perfect consistency* (i.e., if values of D_i and D_j are desired at t_j, we would like to provide values that co-occurred precisely at t_j). To satisfy these ideal conditions it is necessary for our system to be able to record updates at nearly every instant of time, *while* ensuring that transactions are processed in a timely fashion as well. Given the large state spaces of our target control environments (e.g., a telecommunications database may contain a million objects), and current processing capacities, such ideal requirements are unlikely to be fulfilled any time in the near future. Thus, while recognizing that perfect recency and consistency are unachievable, it is our goal to provide *reasonable* recency and temporal consistency. Simply stated, reasonableness means: (a) if value of D_i is desired at t_1, and it is not possible to perfectly satisfy this request, we would provide a value at a time t_2, such that t_2 was *close* to t_1, and (b) if values of D_i and D_j were desired at t_1, and perfect temporal consistency was unsatisfiable, we provide a value of D_i at t_2 and a value of D_j at t_3, such that t_2 and t_3 were reasonably close to each other, while, at the same time, both t_2 and t_3 were reasonably close to t_1. Note the preceding discussion was deliberately left rather imprecise to communicate the basic theme of what we want to do.

We use a measure called *Maximum Tolerable Delay* (MTD) to quantify *reasonable closeness* referred to in the previous paragraph. Using this parameter allows developers of different applications to set their own level of tolerance. Using this measure we define the notions of *recency* and *temporal consistency* used in this paper.

Definition 5 *Version v of data item D_i, i.e., D_i^v is considered to be* **recent** *with respect to time t_j if and only if* $|\mathrm{DTS}(D_i^v) - t_j| \leq \mathrm{MTD}$.

Definition 6 *D_i^v and D_j^w are considered to be* **temporally consistent** *if and only if* $|DTS(D_i^v) - DTS(D_j^w)| \leq \mathrm{MTD}$.

Note that we use absolute values in these definitions. The reason is that the retrieved data versions could have timestamps which may be *smaller* or *larger* than the request timestamp. This is illustrated through the following example. Consider a transaction T_i that wishes to read D_j. Further assume that two versions of D_j, D_j^v and D_j^w exist such that $DTS(D_j^v) \leq RQTS(T_i) \leq DTS(D_j^w)$. Moreover, $RQTS(T_i) - DTS(D_j^v) > MTD$ and $DTS(D_j^w) - RQTS(T_i) < MTD$. In such a situation, we contend that the transaction should be provided the value of D_j^w, even though this version occurs *after* the request timestamp of T_i. Our rationale is that by virtue of being within MTD time units, D_j^w offers a truer view of D_i with respect to $RQTS(T_i)$ than does D_j^v. These assumptions could be changed without affecting in any way the subsequent work presented in this paper [1].

Another interesting fact to note is that definitions 5 and 6 mandate that data item values be updated at least once every MTD units. Otherwise, recency and temporal consistency may be unsatisfiable. In a periodic update arrival scenario, this is easily achieved by requiring that all sensors have periods less than MTD. In this paper however, we examine an aperiodic update arrival case, where it is not as simple as the periodic case. As mentioned previously, in the aperiodic case, data values are typically reported when they change. However, if an object state remains constant for a time period longer than MTD, the corresponding data value in the database will lose its recency (even though it represents the "correct" state of the object). Clearly, some mechanism is required to handle this anomaly. In this paper, we make the assumption that no value in the system will go unreported for longer than MTD units of time. In other words, even if a value does not change, it *will still be reported* at least once every MTD units of time. In the experiments reported later on in the paper, this constraint is enforced for every data item. Thus, recency is always satisfiable in our system.

When a transaction wishes to read a number of data items, the values returned must be *both* recent and temporally consistent. Satisfying both properties simultaneously requires the concurrent satisfaction of the conditions specified in definitions 5 and 6 above.

Definition 7 *A set of data item versions $\mathcal{D}$, is considered to be* **recent** *with respect to a time t_i and* **temporally consistent** *if and only if the following two conditions are simultaneously satisfied:*

1. $\forall D_i^v \in \mathcal{D}, |\mathrm{DTS}(D_i^v) - t_i| \leq \mathrm{MTD}$;
2. $\forall D_i^v \in \mathcal{D}, \forall D_j^w \in \mathcal{D}, |\mathrm{DTS}(D_i^v) - \mathrm{DTS}(D_j^w)| \leq \mathrm{MTD}$.

Next, we turn our attention to designing mechanisms to attempt the satisfaction of the above rules. The key factor that will influence the degree to which we satisfy recency and temporal consistency is how well we are able to install updates. If we can install updates well, transactions will read "good" values. We consider the "goodness" of the update installation procedure to be dependent on two interrelated factors.

1. How *efficiently* are updates installed? Efficient update installation would mean that data items are updated nearly as rapidly as updates arrive at the system. If the update installation process is inefficient (i.e., the system is not able to install a large number of updates), transactions will read non-recent values.

2. How *fairly* are updates installed? Given that a large number of updates are arriving rapidly and unpredictably, the system must ensure that it installs values of *all* data items and not *some* data items. If the update installation process is unfair, transactions will retrieve temporally inconsistent values.

[1]Readers familiar with [38] will note similarities between the above notions and those presented in [38]. While we encourage interested parties to consult [38], we emphasize that our definitions are semantically richer. Also, our use of a single validity interval, MTD, yields a much simpler model which matches actual application domain requirements (e.g., [9]).

In systems such as those being considered in this paper, update patterns could be highly random and completely unpredictable. The notion of fairness (item (2) above) is an important one in such scenarios. To see this, consider a database with two data items D_i and D_j. An efficient but unfair installation procedure may update D_i very frequently but D_j very seldom. A fair installation policy on the other hand, would ensure that both data items are updated on a regular basis. Clearly, transactions that read updates of the second installation procedure would be expected to perform better control actions than those that read the output of the first policy. Below we describe an efficient and fair update installation policy that we have designed.

4.1 Update Installation: Efficiency and Fairness

Before stating the installation algorithms we describe a structure called *TS_ARRAY* (Timestamp Array) that is utilized by the algorithms. *TS_ARRAY* is a memory resident array of dimension *NumSDItems* which denotes the number of sensor data items in the database. The *TS_ARRAY* stores the largest timestamp of each sensor data item, i.e., the data timestamp of the most recently created version of each data item. More precisely,

$$TS\_ARRAY[i] = DTS(D_i^v)$$

where D_i^v is the most recent version of D_i, i.e., the version with the largest timestamp.

As mentioned before, the goal of our update installation policy is to offer efficiency as well as fairness. Let us first consider efficiency, i.e., ways to maximize the number of installed updates while ensuring that system resources remain available for transaction processing. We recognize that in any real system, given that large number of updates as well as transactions arrive within short time intervals, it is going to be very difficult to install every update. Thus at high update arrival rates, it may be beneficial to discard some updates while ensuring that recency is preserved. We attempt to achieve this using the following basic philosophy: we attempt to install updates such that recency is only preserved beyond some small interval *RI* (Recency Interval), $0 \leq RI \leq MTD$. In other words, we only attempt to install updates such that data items are updated (i.e., newer versions are created) at most once every *RI* time units but at least once every *MTD* time units. Were this goal to be achieved, then the system could guarantee that it can fulfill any data request within at least *RI* time units and at most *MTD* time units of the request timestamp. To ensure a high level of recency in the system while allowing for transaction processing, the system increases *RI* such that the update processing load decreases when the miss ratio (MR) of transactions reaches unacceptably high levels. If, on the other hand, MR is very low, then *RI* is decreased, thereby attempting to maintain higher levels of recency in the system by giving updates a greater share of system resources. These adjustments of *RI* are achieved through a procedure that adapts dynamically based on system feedback. Details are provided below.

To properly appreciate our procedure for installing updates, it would be helpful to first understand our update application model. As mentioned before, we assume an aperiodic update arrival scenario, i.e., updates arrive when the state of real-world entities changes. Each update changes exactly one data item. More precisely, the result of each installed update is the creation of a new version of the relevant data item. However, in order to enforce our recency rule (given in definition 5), we impose the condition that even if the value of a data item does not change, two consecutive updates to any data item can be at most *MTD* units of time apart. This condition, although admittedly artificial, allows us to evaluate how good our algorithms are in maintaining recency. It is easily seen that a perfectly efficient update installation procedure (i.e., one that installs every update) will never allow the system to become non-recent. This is so because the abovementioned condition guarantees that each data item will receive at least one update every *MTD* time units. The conceptual basis of this condition is very closely related to the notion of *update patterns* identified in [29]. When an update arrives, it is handled in one of two ways, depending upon our update process model, discussed below.

The authors argued convincingly in [3] that updates are best handled by a single process (as in CICS), as opposed to each update being handled as a separate transaction. It must be kept in mind however, that in [3], the authors were dealing with a main-memory database system. In a disk-resident database, as is our case, it is not clear that a single update process is the best way to go. This is because, in the disk-resident case, delays at disk may be large enough to make concurrency among updates attractive. To examine this issue, we consider two different process models: *Uni-Process Update Model* (UPUM) and *Multi-Transaction Update Model* (MTUM). In UPUM, updates are applied as a single update installation process, whereas in MTUM, each update is considered as a separate transaction. Whereas MTUM is easily understood, UPUM requires a little more explanation.

Under UPUM, the update process installs updates from an *update queue*. The essence of the process is as follows: when an update arrives at the system, it is enqueued within the database system. This queue is maintained *not in the order of arrival* but according to the level of recency of the target data item. More precisely, an update $U(D_i)$ (i.e., update to data item i) would be enqueued by checking the value of *TS_ARRAY[i]*. Lower the value of *TS_ARRAY(i)*, the older D_i is, and the more urgent it is to install that particular update in order to make D_i recent, and consequently raise the position of the corresponding update in the update queue. This queueing order is very significant as it enforces *fairness* in our system. As a data item loses its recency, updates to it gain better and better position in the update queue, ensuring that all data objects get equal chance of being recent. Finally, the update process installs updates from this queue.

Modeling updates as UPUM or MTUM raises some interesting issues which have performance implications. These issues concern the overhead and efficiency tradeoffs in the models, e.g., in MTUM, because of the large number of updates that arrive, potentially a substantial overhead (e.g., context switching) may be associated with scheduling these. In UPUM, such overheads are not present. However MTUM has the potential of deriving performance gains, by exploiting concurrency (in UPUM, only one update may be active at any one time). Thus, our goal in studying these two models was to examine the effects of the above issues on system performance.

Now we describe the procedures that implement the update installation mechanism. Note that these procedures apply to both UPUM and MTUM.

Procedure RECINT

Adaptive Feedback Procedure To adjust Recency Interval (*RI*). This procedure assumes the existence of a dynamic control variable α (explained later) and a parameter called *SampleBatch*.

```
After every SampleBatch transactions leave the system
  if $MR >$ ThresholdMR
      then $\alpha \leftarrow \min(1.0,\ \alpha \times 1.10)$;
      else $\alpha \leftarrow \alpha \times 0.90$;
  $RI \leftarrow \alpha \times MTD$;
```

Procedure UINSTALL

The actual update installation procedure.

```
On arrival of update $U(D_i)$ meant for data item $D_i$
  if $TS(U(D_i)) - TS\_ARRAY[i] \geq RI$      /* $D_i$ has not been updated recently */
      then install $U(D_i)$;
```

```
        else abort $U(D_i)$;
```

The essence of the update installation process is as follows: when an individual update comes up for processing (in either of the two models, i.e., UPUM or MTUM), the UINSTALL procedure is performed to decide whether to actually commit resources to install the update or discard the update (thereby enabling the system to allocate resources to other processes). However, the decision to discard should not lead to recency violation of the concerned data item. This decision is made using the *recency interval* (RI), which is adaptively updated using the RECINT procedure.

We first define the *recency interval* (RI) as a fraction of MTD. More precisely, we define *RI* as: $RI = \alpha \times \text{MTD}$ where α is a dynamic control variable in the range $0 < \alpha \leq 1$ (α is initialized to .20 at startup). When α is very small, so is *RI*, and when α is 1, $RI = MTD$. The value of *RI* directly impacts our processing strategy for updates as can be seen from procedure UINSTALL. This procedure outlines a strategy by which we choose to discard some updates based on the recency of the target data item. As explained above, the logic behind this is that by discarding certain updates, we free up resources for use by transactions. Our goal is to install at most one update per data item every *RI* time units. This means that if *RI* is small, we will attempt to install updates at a very rapid rate (i.e., discard fewer updates). On the other hand, if *RI* is large, we will attempt to install updates at a slower rate and succeed in discarding a larger number. We compute a system characteristic *Deadline based Miss Ratio* (DMR) as:

$$DMR = \frac{\textit{Number of transactions missed because of deadline misses}}{\textit{Total number of transactions entering the system}}$$

The notion of *DMR* is introduced because in these systems, transactions misses occur due to two reasons, (a) deadline misses (quantified by *DMR*); and (b) reading of temporally inconsistent values (quantified by a characteristic *TCMR* introduced later in section 4.2). As will be explained later, *TCMR* is not strongly related to the system load, and thus should not be factored into load dependent decisions. On the other hand, *DMR* is the true indicator of the load on the system. We have formulated the maximum allowable *DMR* of transactions as an application-dependent parameter *ThresholdDMR*. When *DMR* exceeds *ThresholdDMR*, we need to devote more resources to transaction processing and consequently increase α, thereby expanding *RI*, which enables the system to discard more updates. This frees up system resources for transaction processing. In other words we sacrifice recency to achieve lower miss ratios. The adaptive recomputation of α is performed every *SampleBatch* (another parameter) transaction completions to capture the recent behavior of the system. Below we provide an illustrative example of the update installation procedure. **Example:** Consider two updates $U(D_i)$ and $U(D_j)$ that arrive at the system with identical timestamps[2] of 50. Assume current time CT = 55. Further assume that D_i and D_j were last updated by updates with timestamps 40 and 30 respectively, i.e., *TS_ARRAY[i]* = 40 and *TS_ARRAY[j]* = 30 respectively. Assume $MTD = 35$ and $\alpha = .5$.

Case 1 (UPUM): In the UPUM case, the two updates are first enqueued in the update queue. As, mentioned before, the queueing order is *age* of the target data items. In this example D_j is older that D_i, as *TS_ARRAY[j]* < *TS_ARRAY[i]*. Thus, $U(D_j)$ would be queued before $U(D_i)$. Now, the current recency interval (RI) of the system is given by, $RI = \alpha \times MTD = 0.5 \times 35 = 17.5$. This means that at this time the system's goal is to install at most one update per data item every 17.5 time units. Thus when the update process chooses $U(D_j)$ for service from the update queue ($U(D_j)$

[2]This can happen as we are modeling a system where distributed sensors are sending updates which travel over a network and arrive at the database. We assume that the sensor clocks are synchronized.

comes up for service earlier than $U(D_i)$ owing to its higher position in the queue), it first checks whether to commit resources to install this update at all by performing the test in UINSTALL. This test computes, $\mathit{TS}(U(D_j)) - \mathit{TS\_ARRAY[j]} = 50 - 30 = 20 > \mathit{RI}$. Thus, the update process would actually install $U(D_j)$. However, when $U(D_i)$ reaches the head of the update queue, it would be discarded (i.e., not installed) as it would not satisfy the test in UINSTALL.

Case 2 (MTUM): In this model, the two updates are modeled as different transactions and thus compete for resources unlike the UPUM case. Here they are treated as transactions and are awarded priorities. Priority assignments are treated in Section 5. Suffice it to say here that by virtue of the fact that $\mathit{TS\_ARRAY[j]} < \mathit{TS\_ARRAY[i]}$, $U(D_j)$ would be awarded a higher priority than $U(D_i)$. This would ensure fairness. When the updates actually came up for service the UINSTALL procedure test would be performed to determine which should be installed. Note that the install/discard decision is identical to the one in the UPUM case.

4.2 Retrieval: Ensuring Temporal Consistency

Having described how updates are installed, we now turn our attention to how transactions actually retrieve data. We hasten to add that update installation and retrieval are highly interrelated processes. Basically, transactions can only retrieve what has been installed. However, the reader can very well imagine that from the several data item versions that are potentially available, only certain particular versions need to be retrieved. In this section we elaborate on the retrieval mechanism to fetch versions that best satisfy transaction requirements. Note that our discussion concentrates on sensor data. Non-sensor data, by virtue of being updated slowly, pose little retrieval problems. Thus, in the future discussion whenever we refer to "data items", we are alluding to "sensor data items".

We will illustrate the retrieval procedure with an example: Consider a transaction T that wants to retrieve data items D_i, D_j and D_k in that order. Further assume $\mathit{RQTS(T)} = t_r$. The first operation to be executed would be *read(D_i)*. This operation would select version v of data item D_i, i.e., D_i^v, such that there exists no other version of D_i with a timestamp value closer to t_r. In other words, D_i^v is the *most recent* value of D_i with respect to t_r. Having selected D_i^v, we then select the appropriate versions of D_j and D_k, such that temporal consistency conditions are satisfied between D_i, D_j and D_k. There are two subtle points to note here:

1. Assume that the final result of retrieval for the above transaction is D_i^v, D_j^w, D_k^z. D_i^v was chosen by virtue of being the version of D_i most recent to t_r. Now, when D_j^w and D_k^z are chosen, the overriding criterion is that they should be temporally consistent with D_i^v and with each other. Thus, these versions may not be the closest versions of data items D_j and D_k to t_r. Note that *if temporally consistent data items are not found, the transaction must abort, as temporal consistency is a mandatory condition in our transaction processing model.*

2. In order to satisfy recency the following condition needs to be satisfied as well: (a) $|D_i^v - t_r| \leq \mathit{MTD}$, (b) $|D_j^w - t_r| \leq \mathit{MTD}$, and (c) $|D_k^z - t_r| \leq \mathit{MTD}$, i.e., all the retrieved versions should satisfy recency with respect to t_r. However, if, for some reason, no updates were installed in that period, this condition may be unsatisfiable. In this case what to do with the transaction is really up to the application designer as some applications may tolerate few recency violations while other applications may tolerate none. In the performance study reported in Section 7 we study two cases: (a) transactions reading non-recent data are aborted, and (b) transactions reading non-recent data are allowed to continue.

The above high level discussion outlines the basic requirements of a retrieval policy that seeks to ensure temporal consistency among retrieved data items. A straightforward implementation of such a policy (pairwise comparison between data items) is highly inefficient, requiring $O(N^2)$ time, where N is the number of data items. In real-time scenarios such an implementation will be unacceptable. Below we describe a procedure called *Interval Adjustment Technique* (IAT) that implements our retrieval requirements efficiently.

The intuitive basis of IAT is as follows. In processing the retrieval requirements of a transaction T, we first define an acceptable timestamp interval based on *RQTS(T)*. Subsequently, based on each read access by T, we modify this timestamp interval by factoring in the data timestamp of the data item that was just read. Also, each data access is performed such that the data timestamp of the accessed data item is within the current acceptable timestamp interval. If such data items can be found for the entire read set of T, then temporal consistency is ensured and T completes successfully. Otherwise T is aborted. A procedural description of IAT follows. We denote the *Acceptable Timestamp Interval* of T by the notation ATI_T.

Procedure IAT

This procedure attempts to ensure temporal consistency across the read set of a transaction. The following pseudo-code is written with respect to a transaction T with request timestamp *RQTS(T)*.

```
$ATI_T \leftarrow [RQTS(T) - ATIFactor \times MTD,\ RQTS(T) + ATIFactor \times MTD]$;
                                                            /* Initialize $ATI_T$ */
foreach read access request for data item $D_i$ do
    if no version of $D_i$ exists with timestamp in the range given by $ATI_T$
        then abort $T$;                             /* Temporal Consistency Violation */
        else
            retrieve the version of $D_i$, $D_i^v$, with timestamp closest to the
                                                            midpoint of $ATI_T$;
            $ATI_T \leftarrow ATI_T \cap [DTS(D_i^v) - MTD,\ DTS(D_i^v) + MTD]$;      /* Adjust $ATI_T$ */
```

The IAT procedure is simple: it first initializes the *ATI* of T to a large interval (characterized by the parameter *ATIFactor*) centered on the request timestamp of T. Then, with each data access request, if it is not possible to find a version of the data item in the ATI_T interval, then T must abort as temporal consistency cannot be preserved. If such versions are found, an appropriate version is retrieved and ATI_T is adjusted to factor in the data timestamp of the recently accessed version. The version closest to the midpoint of ATI_T is retrieved to ensure that the largest possible ATI_T results after the adjustment. Other retrieval rules (such as version closest to *RQTS(T)*) will work as well, at the cost of perhaps overly restricting the acceptable interval to a point where feasible versions may exist, but will not be found. The following example illustrates how the above procedure works. **Example:** Consider a transaction T with a request timestamp of 100 which wants to read data items D_i, D_j and D_k, in that order. Table 2 shows different versions of the data items that exist and their associated data timestamps. Assume $MTD = 15$ and $ATIFactor = 5$. ATI_T will be initialized to [25, 175]. When T requests access to D_i, procedure IAT will fetch D_i^r by virtue of its timestamp being closest to 100. Following this access, ATI_T will be adjusted as follows:

$$ATI_T \leftarrow ATI_T \cap [DTS(D_i^v) - MTD, DTS(D_i^v) + MTD] = [25, 175] \cap [75, 105] = [75, 105]$$

Then access to D_j will be requested and D_j^t will be retrieved as its timestamp is closest to the midpoint of the interval [75,105]. Following this access ATI_T will be adjusted as follows:

$$ATI_T \leftarrow [75, 105] \cap [70, 100] = [75, 100]$$

Data Item	**Existing Versions**	**Data Timestamps**
D_i	D_i^p	60
	D_i^q	85
	D_i^r	90
D_j	D_j^s	65
	D_j^t	85
	D_j^u	108
D_k	D_k^v	57
	D_k^w	89
	D_k^x	98

Table 2: Existing Versions and Associated Data Timestamps

Subsequently, access to D_k will be requested and D_k^w will be retrieved. The final result is significant – the versions that were fetched satisfy temporal consistency (and coincidentally, recency as well). However, as can be easily observed, the most recent versions of D_j and D_k with respect to *RQTS(T)* were not fetched.

Note that our retrieval mechanism requires fast and efficient lookup and access of data versions. We assume the existence of an underlying versioning and version indexing scheme. A large number of versioning and access schemes for temporal data have been discussed in the literature (see [41] for an excellent survey and a near-exhaustive list of references). As far as efficient version lookup and access is concerned we are interested in efficient indexing mechanisms geared towards supporting retrievals such as ``‘‘find the version of`` D_i ``closest to time = 100’’``. A very nice scheme for such indexing has recently appeared in [47]. This paper presents a practical and asymptotically optimal time indexing structure for a versioned, timestamped, temporal databases. This structure is shown to optimally solve both snapshot as well as time range queries, with respect to current as well as historical versions. In the simulation experiments described in Section 7 we assume the presence of such a structure.

Finally, in this section we talk about an issue that we alluded to in section 4.1, namely that of transaction misses due to temporal consistency violations. This is something that renders systems such as those considered in this paper not directly amenable to analysis by considering the traditional *Miss Ratio* (MR) metric. In traditional real-time systems, transaction misses are due to resource contention and eventual deadline violations. Thus the pure deadline based MR metric is a very useful one to characterize such systems. In this paper however, we have to consider two types of miss ratios:

1. *Deadline based Miss Ratio* (DMR), defined and described in section 4.1; and
2. *Temporal Consistency based Miss Ratio* (TCMR), defined as the fraction of transactions missed because of temporal consistency violations.

The reader can easily see that while *DMR* is an indicator of system load, *TCMR* is an indicator of the level of efficiency and fairness in the system with respect to updates, i.e., a fair update installation policy will reduce the likelihood of temporally inconsistent retrievals as compared to an unfair installation policy.

So far, in this section, we have described how recency and temporal consistency are maintained through judicious update installation and retrieval methods. In the class of database systems under consideration in this paper, there is another, very important component: real-time processing. We now turn our attention to that.

5 Real-Time Scheduling

In this section we look at a number of algorithms that attempt to schedule transactions and updates by allocating system resources to ensure transactions finish within their deadlines and updates are installed rapidly enough to maintain recency. Note that even though our update installation procedure was described in the previous section, the system still has to decide when to schedule transactions and when to schedule updates. We describe three algorithms to make this decision: (a) Virtual Deadline of Updates (VDU), (b) Updates First (UF), and (c) Transactions First (TF). Whereas VDU is the algorithm that we have developed, UF and TF are baseline algorithms suggested in [3].

5.1 Virtual Deadline of Updates (VDU)

VDU is a priority based scheduling policy, where priority assignments are based on the *earliest deadline* (ED) principle [35]. Under ED, priority assignment to transactions is simple, as transactions arrive with explicit deadlines. The update process (in UPUM) or update transactions (in MTUM) have no explicit deadlines. Thus, we have to devise a priority assignment mechanism for updates that will let the scheduler deal with both workload types (i.e., transactions and updates) uniformly. In VDU a sequence of *virtual deadlines* are assigned to the update process in UPUM, and unique virtual deadlines are assigned to update transactions in MTUM.

5.1.1 UPUM

In UPUM, updates are installed by a single update installation process according to when the target data item was last updated. This order is determined by using values in the *TS_ARRAY*. The problem is as follows: what should the priority of the update process be with respect to the transactions which have clear cut priorities based on their deadlines. We first provide an intuitive answer to this question and then delve into specifics.

The basic idea behind priority assignment to the update process is as follows: a relatively high priority should be assigned to the update process when data values are old, i.e., there is a lack of recency in the database. When there is no perceived problem with recency, the priority of the update process should be relatively small, such that the system can process transactions at a rapid rate. The major issue then is to recognize that there is a problem regarding recency in the database. When values in the database are old, transactions retrieve non recent data. We call such reads *stale reads*. More precisely, an operation *read*(D_i) with a request timestamp of t_r is considered a stale read if the version of D_i returned, D_i^v, fails to satisfy: $|DTS(D_i^v) - t_r| \leq MTD$. Thus, if a large fraction of reads (note that by reads we allude to "sensor data reads") are stale, then one may conclude that there is a problem with recency in the database. In that case the update process needs to be upwardly prioritized to make the database more recent. On the other hand, if the priority of the update process is too high, then the system would be processing updates at the expense of transactions, thereby lowering the capability of the system to meet transaction deadlines. This would be manifested by a high deadline based miss ratio (DMR) in the system. Thus, assigning priority to the update process, is *balancing the conflicting objectives of having a low number of stale reads (i.e., having good database recency), while achieving low miss ratios.*

Now we describe how the above goals are achieved in the VDU algorithm. As mentioned before, the basic priority assignment philosophy used is ED. However, the update process does not have an explicit deadline. The VDU policy awards a sequence of *virtual deadlines* (VD) to the update process to control its priority with respect to the transaction processes. More precisely, the update process is awarded a *virtual deadline* of $VD = Current\_Time + \beta \times MTD$. β is a dynamic control variable defined in the range, $0 \leq \beta \leq 1.0$. Thus, the maximum value of VD is $Current\_Time + MTD$ (when $\beta = 1.0$). This deadline is assigned with the goal of scheduling at least one update within the next MTD time units, as the system is sure to lose recency if not even one single update is scheduled in that period.

The above discussion points out that the update process has a *sliding* priority, i.e., the update process is assigned virtual deadlines that slide between $Current\_Time$ and $Current\_Time + MTD$. Now we address the issue of how this deadline assignment is actually achieved. At system startup (i.e., time = 0), $VD = MTD$. The intention of this initial assignment is to have the update process serviced within MTD time units. Each time the update process experiences a CPU service, the virtual deadline is reset as follows: $VD \leftarrow Current\_Time + \beta \times MTD$, thereby varying the priority of this process.

The dynamic variable β adapts based on system feedback, and represents how the VDU algorithms achieves the balance identified above, i.e., the balance between ensuring few stale reads and low miss ratios. To understand how this happens, we first define a system characteristic *fraction stale reads* (FSR), which is computed as:

$$\text{FSR} = \frac{\textit{number of stale reads}}{\textit{total number of sensor data reads}}$$

After every *SampleBatch* transactions leave the system we perform two tests and correspondingly adjust β (β is set to 0.25 at system startup). The tests and corresponding adjustments may be expressed through the following decision rules **R1** and **R2**:

```
Decision Rule R1
  Adaptation of $\beta$ based on stale reads.
 if $FSR >$ ThresholdFSR
     then $\beta \leftarrow \beta \times 0.95$
     else $\beta \leftarrow \beta \times 1.05$
Decision Rule R2
  Adaptation of $\beta$ based on miss ratio.
 if $DMR >$ ThresholdDMR
     then $\beta \leftarrow \beta \times 1.05$
     else $\beta \leftarrow \beta \times 0.95$
```

The two decision rules are easy to understand. **R1** says that if FSR is greater than a threshold FSR (denoted by the parameter *ThresholdFSR* which is set to 0.30 in all our experiments) that indicates recency is low. In that case β is decreased, which lowers the VD, thereby increasing, comparatively, the priority of the update process. On the other hand if system recency is acceptable, the reverse actions are performed. Similarly, **R2** adapts β based on DMR.

5.1.2 MTUM

Next we turn our attention to how the VDU algorithm works in the multitransaction update model. Here the algorithm is relatively straightforward compared to the UPUM case. In MTUM, each update is considered a separate transaction and is assigned individual virtual deadlines. The deadline assignment policy is as follows: consider an update $U(D_i)$. Let us use the notation VD_i to refer to the virtual deadline of $U(D_i)$. The deadline assignment is done according to the following procedure.

```
Procedure ASSIGN_VD
  Procedure to assign $VD_i$ to update $U(D_i)$ in MTUM.
 if ($TS\_ARRAY[i] + \beta \times MTD$) > $Current\_Time$
    then $VD_i \leftarrow TS\_ARRAY[i] + \beta \times MTD$;
    else if ($TS\_ARRAY[i] + MTD$) > $Current\_Time$
       then $VD_i \leftarrow TS\_ARRAY[i] + MTD$;
       else $VD_i \leftarrow Current\_Time + \Delta$;
```

Basically, the above procedure assigns a virtual deadline to updates $U(D_i)$ based on how long it has been since the last update of D_i. β is a dynamic control variable, which changes in precisely the same way as in the UPUM case, i.e., based on the decision rules **R1** and **R2**. Thus, depending upon *FSR* and *DMR*, updates are given higher or lower priorities with respect to transactions. Note however, that β has no effect on the relative priority of updates with respect to other updates. Δ is a system parameter.

5.2 Other Scheduling Mechanisms

We compare the performance of VDU with two other algorithms discussed in [3], *Updates First* (UF) and *Transactions First* (TF). For these two algorithms we consider both the UPUM as well as the MTUM case.

UF: In UF, all updates are awarded higher priorities than all transactions. A running transaction is preempted upon arrival of an update. In the UPUM case, updates are queued in FIFO order, whereas in MTUM, updates are processed in *earliest-arrival-first* order.

TF: In this method all transactions have higher priority than all updates. Updates only get installed when there are no transactions in the system. Since updates are small duration jobs, transactions do not preempt updates. Again, as in UF, in the UPUM case, updates are queued in FIFO order, whereas in MTUM, updates are processed in *earliest-arrival-first* order.

For both the above algorithms we apply the temporal consistency enforcing retrieval mechanism described in section 4.2. This makes the “playing field” even for all three algorithms (i.e., VDU, UF and TF).

In all three algorithms described so far (i.e., VDU, UF and TF) a transactions reads data items existing in the database. In [3] the authors describe and evaluate another smart algorithm called *On Demand* (OD). In OD, if a transaction wishes to read the most recent values and encounters a stale value in the database, an attempt will be made to get and install a fresh update from the update queue in the UPUM case. OD is an extension of TF – transactions still have higher priority over updates. OD requires significantly more knowledge than the other algorithms. It assumes that queued updates can be searched *and* the right update identified. We wanted to consider OD as well in our experiments. However, OD is impractical in our case, as it is not possible to adapt OD to the requirement of maintaining temporal consistency. In OD a transaction extracts and installs the most fresh (i.e., recent) update from the update queue. However, what happens if this update is not temporally consistent with previously read values? In fact as shown in the example in section 4.2, the most recent update need not correspond to temporal consistency. Thus a transaction will not only have to look for a recent update it will have to look for a simultaneously temporally consistent one. This of course will be prohibitively expensive. Also, as can be easily seen, this gets even more complicated in the MTUM case.

6 Simulation Model and Performance Metrics

6.1 Simulation Model

To evaluate the performance of the update installation and scheduling algorithms outlined before, we conducted extensive simulation experiments. Our simulation programs were written in SIMPACK, a C/C++ based simulation toolkit [16, 17].

The real-time, temporal database system consists of a shared-memory multiprocessor that operates on disk-resident data. We model the database as a collection of pages. A transaction is modeled as a sequence of read and write page accesses, while an update is a singleton write, i.e., it is a write request to a page. A read request submits a data access request to the concurrency control (CC) manager, on the approval of which a disk I/O is performed to fetch the page into memory followed by CPU usage to process the page. Similar treatment is accorded to write requests with the exception that write I/Os are deferred until commit time. Thus, processing an update involves first reading the appropriate page and then writing the modified page back.

Our RTDBS model is shown in figure 2. There are four major components:

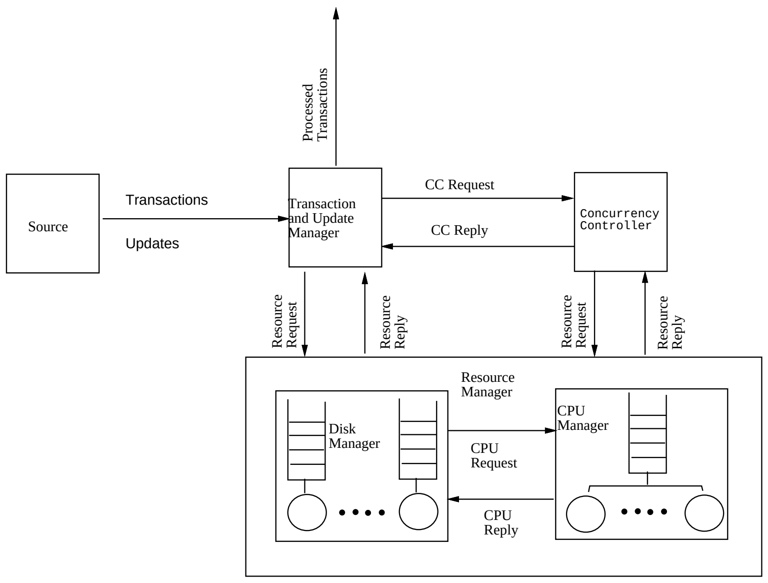

Figure 2: The RTDBS Model

1. An *arrival generator*, that generates the real time workload with deadlines
2. A *transaction manager* that models transaction and update execution and implements the scheduling algorithm
3. A *concurrency controller*, that implements the CC algorithm, in our case OPT-BC [36]

4. A *resource manager* that models system resources, i.e., CPUs and disks and the associated queues

6.2 Database Model

Our database model consists of a number of pages denoted by the parameter *DBSize* and a parameter *ItemsPerPage* that denotes the number of data items per page. We assume a dichotomy of storage between sensor and non-sensor data items, i.e., no database page stores both sensor and non-sensor data items. A parameter *SensorFrac* denotes the fraction of the *DBSize* pages that store sensor data items. Thus the number of sensor data items is given by:

$$NumSDItems = DBSize \times SensorFrac \times ItemsPerPage$$

The purpose of defining the *NumSDItems* parameter will be clear from section 6.4.1 below.

6.3 Resource Model

Our resource model considers multiple CPUs and disks as the physical resources. For our simulations we assumed the data items are uniformly distributed across all disks. The *NumCPU* and *NumDisk* parameters specify the system composition, i.e., the number of each type of system resource. The parameter *DBSize* denotes the number of data pages, while a parameter *SensorFrac* is used to denote the fraction of data pages that correspond to sensor data. The sensor data items themselves are assumed to be uniformly distributed across disks. There is a single queue for the CPUs and the service discipline is assumed to be preemptive-resume based on the task (i.e., transactions or updates) priorities. Each individual disk has its own queue with a non-preemptive, transaction priority based service discipline. The parameters *ProcCPU* and *ProcDisk* denote the CPU and disk processing time per page respectively. To accurately model the UPUM-MTUM tradeoff we model the context switch cost through a parameter *ContextSwitch*. Whenever there is a context switch, *ContextSwitch* amount of time is charged to the process that is *switched in*. These parameters are summarized in table 3.

6.4 Workload Model

Our workload model consists of modeling the characteristics of updates and transactions that arrive and are processed in the system as well as their arrival rate.

6.4.1 Updates

As mentioned before in Section 3, our system assumes aperiodic updates with the guarantee that two consecutive updates to the same data item are never more than *MTD* time units apart. This condition was imposed to make recency always satisfiable – thus, based on the level of recency violations, it would be possible to test the effectiveness of algorithms. Each sensor data item is modeled as a generator of updates i.e., each sensor data item D_i generates a stream of *write*(D_i) updates that arrive immediately at the system. In other words, for a given data item, the generation pattern of updates is equivalent to the arrival pattern of those updates at the database. The actual arrival pattern of all updates at the database, thus, is the convolution of the generation patterns of each D_i. All clocks are assumed to be synchronized. Each sensor data item generates updates with inter-generation times distributed in a triangular fashion. More specifically, the inter-generation times of updates from a given sensor data item D_i are generated from a triangular distribution with left bound 0, right bound MTD, and mode M_i (the reason for choosing a triangular distribution is elaborated below). For each D_i, the mode of inter-generation times M_i is chosen from a truncated normal distribution with mean $\frac{MTD}{UpdateRate}$. The truncation is such that the generated mode is always in the range $[0, MTD]$.

From the above description, it is easy to estimate the approximate update arrival rate into the system. Clearly, the expected frequency of update arrivals, $\mathcal{U}$, is given by,

$$\mathcal{U} = \text{Number of sensor data items} \times \frac{1.0}{\text{Average inter-generation time of updates for a sensor data item}} \tag{1}$$

Since we know that inter-generation times are distributed triangularly, the average inter-generation time for a data item is simply the mean of the triangular distribution with left bound 0, right bound MTD, and mode $\frac{MTD}{UpdateRate}$. In other words, equation 1 may be re-specified as

$$\mathcal{U} = NumSDItems \times \frac{1.0}{\left(\frac{0+MTD+\frac{MTD}{UpdateRate}}{3}\right)} \tag{2}$$

Substituing the values of the parameters used in our experiments from table 4, $\mathcal{U}$ works out to approximately 33.75 updates/second. This number represents a reasonably loaded system with regard to updates. The reader, however, is reminded that each update constitutes a singleton write (i.e., read a page, update, write it back). Thus, while 33.75 arrivals per second may appear to be rather substantial, this is somewhat mitigated by the fact that update transaction sizes are small.

Finally, we comment upon our choice of the triangular distribution for the inter-generation times. It would appear, superficially, that the exponential distribution, which is almost universally used to generate inter-arrival times, would have been a better choice. There are three reasons for our choice:

1. We required that no two updates from the same data item be more than MTD units apart, i.e., inter-generation times are strictly restricted to the range [0, MTD]. The triangular distribution offered a very natural way of achieving this by declaring a left bound of 0 and right bound of MTD, and thereby defining a range the [0, MTD]. An exponential distribution (without truncation) cannot offer this guarantee.

2. The second, and more subtle reason, has to do with the memoryless property of distributions. One of the nice things about the exponential distribution is that it is the only continuous distribution with the memoryless property. In our model however, this is undesirable, as we require each data item to "remember" when it generated its most recent update such that the next update may be generated within MTD time units.

3. From the two above reasons it appears that the exponential distribution offers certain problems in our scenario. Thus the question was what inter-generation time distribution should we use? It is generally accepted (see, e.g., [32]) that in the absence of definite knowledge, a triangular distribution is recommended as a "rough" distribution.

6.4.2 Transactions

Transactions arrive at the system as a poisson stream with a mean rate of *ArrivalRate*. We consider three broad classes of transaction characteristics: (a.) $Size_T$, which denotes the number of pages accessed by T; (b.) D_T, the deadline of T; and (c.) $RQTS_T$, the request timestamp of T, i.e., the time at which T desires its observed values to have been valid. The service demand of T is denoted as $SD_T = Size_T \times Proc_T$ (recall that $Proc_T$ is the time required to process a page). The arrival generator module assigns a deadline to each transaction using the following expression: $D_T = SD_T \times SR_T + A_T$, where D_T, SR_T and A_T denote the deadline, slack ratio and arrival time of transaction T respectively. Thus, the time constraint of transaction T, $C_T = D_T - A_T = SR_T \times SD_T$. In other words, SR_T determines the tightness of the deadline of T. Note that transactions were generated such that their working sets corresponded to figure 1. Each transaction also has an associated request timestamp, $RQTS_T$. Note that the request timestamp characterizes the temporal nature of the simulated system. Our workload parameters are summarized in table 3. Table 3 contains some parameters not discussed so far. The value of the *WriteProb* parameter

Parameter Type	**Notation**	**Description**
Resource	*NumCPU*	Number of CPUs
	NumDisk	Number of disks
	ProcCPU	CPU time /data page
	ProcDisk	Disk time/data page
	ContextSwitch	Context switch time, charged to the process switched in
Workload	*ArrivalRate*	Transaction Arrival Rate
	DBSize	Number of pages in the database
	ItemsPerPage	Number of data items per page
	DBSize	Number of pages in the database
	NumSDItems	Number of sensor data items
	SensorFrac	Fraction of sensor data pages
	ArrivalRate	mean arrival rate of transactions
	UpdateRate	Constant arrival rate of updates
	WriteProb	Write probability/accessed page
	SizeInterval	Range of the number of pages accessed per transaction
	SRInterval	Range of slack ratio
	RQTSInterval	Range from which request timestamps are drawn

Table 3: Input Parameters to our RTDBS Model

denotes the probability with which each page that is read will be updated. *SizeInterval* denotes the range within which transaction sizes will uniformly lie. In other words, the arrival generator module generates transaction sizes by drawing from a uniform distribution whose range is specified by *SizeInterval.* Similarly, transaction slacks are generated by drawing from the uniform distribution whose range is specified by *SRInterval.* Request timestamps are generated by drawing from a uniform distribution whose range is specified by *RQTSInterval.*

6.5 Performance Metrics

We are primarily interested in the following measures of performance:

1. How well does our algorithm do with respect to time cognizance? This is measured by the usual

Overall Miss Ratio metric (OMR). In our case however, transactions are missed due to two reasons, temporal consistency violations as well as deadline misses. We are therefore also interested in the *DMR* and *TCMR* metrics defined in sections 4.1 and 4.2 respectively.

2. How far were we able to make sure that transactions read recent values? This is measured using the *Fraction of Stale Reads* (FSR) metric, which was presented in section 5.1.1.

3. Finally, we are interested in what fraction of transactions went through successfully without reading a single stale data item. This is especially important for applications that are highly intolerant to recency violations. This is measured using the *Successful Without Stale Values* (SSV) metric, which is computed as:

$$\text{SSV} = \frac{\text{number of transactions completing within deadline without reading any stale values}}{\text{total number of transactions}}$$

7 Performance Results

We studied the performance of the competing algorithms under various conditions of process models (i.e., UPUM or MTUM) and system resource configurations (number of CPUs and disks). The basic parameter settings for our experiments are shown in Table 4 below.

Parameter	**Value**
MTD	60 sec
$ProcCPU$	10ms
$ProcDisk$	20ms
$ContextSwitch$	1ms
$DBSize$	10000 pages
$SensorFrac$	0.9
$ItemsPerPage$	1
$NumSDItems$	900
$UpdateRate$	3
$WriteProb$	0.25
$SizeInterval$	[1,15]
$SRInterval$	[2,6]
$RQTSInterval$	$[Current\_time - 5 \times MTD,\ Current\_time]$
$ATIFactor$	5
$ThresholdMR$	0.2
$ThresholdFSR$	0.3
Δ	1
$SampleBatch$	100

Table 4: Parameters Settings for Our Experiments

7.1 UPUM

We first discuss the results of the experiments under the UPUM model, i.e., updates are handled as a single process.

7.1.1 High Resource Contention

We first explored a highly resource constrained scenario, by considering a uniprocessor machine. In this set of experiments *NumCPU* was set to 1 and *NumDisk* was set to 2. Figures 3 A, B, C, D and E show the results of these experiments. We first check the *OMRs* of the algorithms by

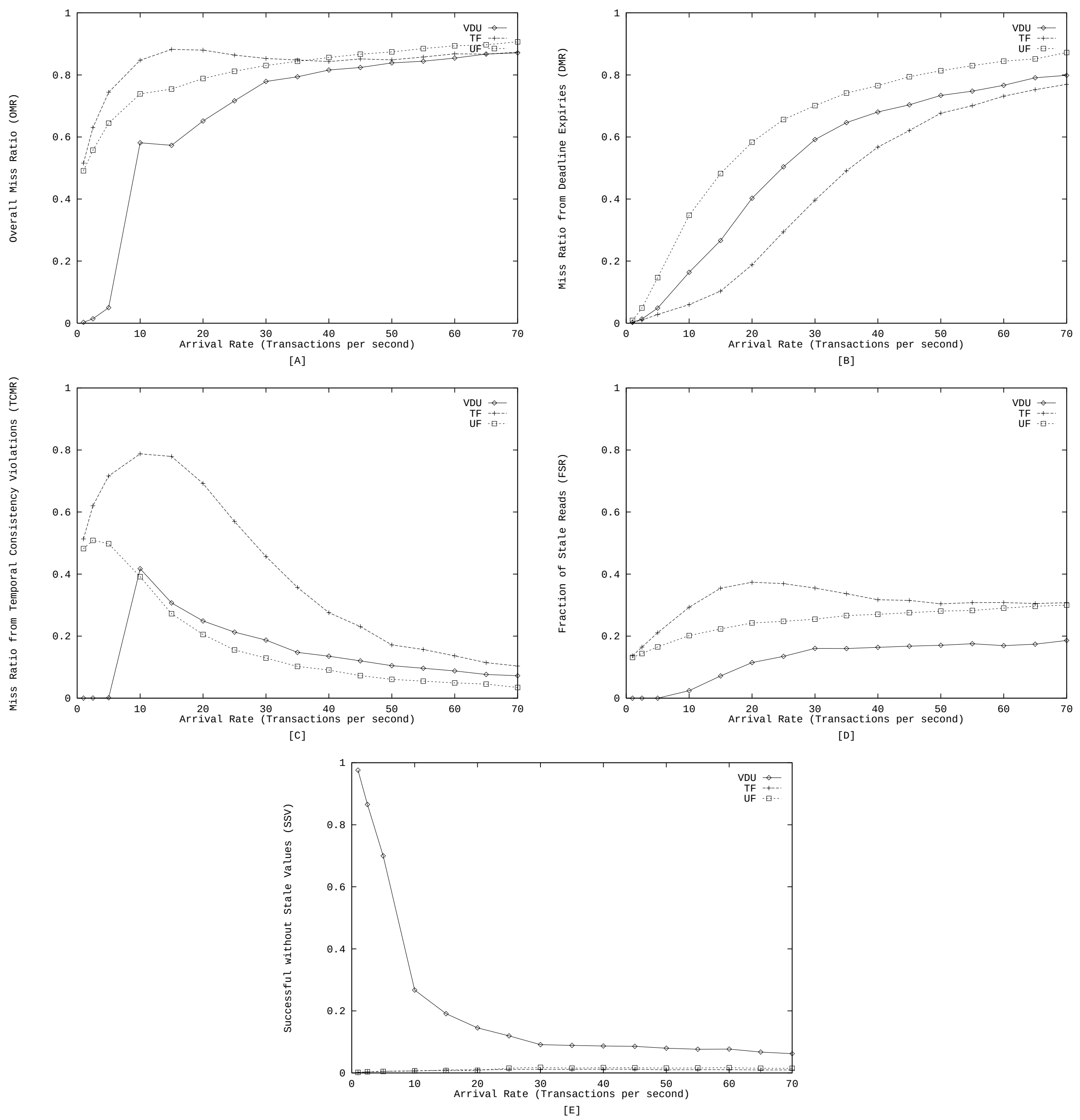

Figure 3: Experiments under UPUM, high resource contention

examining figure 3A, which shows that VDU exhibits the lowest overall miss ratio of the three algorithms. One noteworthy feature of figure 3A is the particular shape of the miss ratio curves.

MR curves typically are S-shaped. The *OMR* curves in figure 3A however, are somewhat different – they rise at the beginning and then exhibit a shallow dip or trough (a small dip for UF and VDU but a long one for TF). Following this dip, the curves rise again. The reason for this apparently strange shape is the fact that transaction misses in these systems happen because of two reasons: deadline expiries and temporal consistency. The precise effect of these two factors is shown in figures 3B and C. Figure 3B plots *DMR* (i.e., the miss ratio due to deadline expiries) against the transaction arrival rate. TF, by virtue of unilaterally awarding higher priority to transactions, shows the lowest *DMR* while UF, by discriminating against transactions, shows the highest. The VDU curve is in between the UF and TF curves as it attempts to balance the priorities of updates and transactions. It is noteworthy that the VDU curve is closer to the TF curve than the UF curve, indicating that VDU adapts well to the changing processing needs of the environment. Note that the *DMR* curves are semantically identical to traditional MR curves and exhibit the same S-shape. Figure 3C plots the *TCMR* curves, i.e., it shows the miss ratios due to temporal consistency violations. A remarkable feature of this figure is the bell-shaped nature of these curves, which is related to the level of resource contention in the system. At low transaction arrival rates, resource contention is relatively less, allowing a large number of transactions to access data items, leading to the increasing trend of the *TCMR* curves. Note that at very low arrival rates, especially for UF and TF, *TCMR* dominates *DMR*. As arrival rates get higher and higher, resource contention increases, which means transactions increasingly spend more time waiting at resource queues, and less time accessing and processing data. Thus, *TCMR* decreases, while *DMR* increases. This is manifested by the dominance of *DMR* over *TCMR* at higher arrival rates. Regarding the *TCMR* curves, as expected, TF exhibits worst behavior by virtue of its discriminating against updates, as a result of which it is able to install fewer updates than the other two algorithms, leading to difficulty in satisfying temporal consistency. Also, UF exhibits the best overall behavior for exactly the opposite reasons. The VDU curve is in the middle. One interesting fact to note is that at very low arrival rates (≤ 10 transactions/sec) VDU has lower *TCMR* than UF. This may seem counterintuitive at first glance – how can UF, which is predisposed towards favoring updates, do worse than VDU, which attempts to balance updates and transactions? The answer to this apparent anomaly lies in the notion of *fairness* discussed in section 4.1. Recall, that the update installation process in VDU is fair as the updates are queued in order of age of target data item. The update installation policy in UF, on the other hand is FIFO and thus not fair. At very low arrival rates, when contention in the system is quite low, VDU and UF perform more or less similarly with respect to the *fraction of total updates* installed (this was measured by our experiments but not reported in this paper for the sake of brevity). However, VDU, by virtue of its inherently fair installation policy, is better able to satisfy temporal consistency. At higher arrival rates however, UF is able to install a larger fraction of arriving updates than VDU and this counterbalances the effect of increased fairness. Finally we are in a position to explain the (suspicious !!) trough in the *OMR* curves of figure 3A. Clearly, $OMR = DMR + TCMR$. The trough represents the point where *TCMR* just starts to fall off steeply but *DMR* has not established its dominance yet. Soon afterwards, however, *DMR* increases rapidly, causing *OMR* to rise again.

Another noteworthy point emerges from figure 3A with respect to the relative positioning of the UF and TF curves. It seems counterintuitive at first glance that UF and TF should have such similar performances at high loads (the curves are very close together), while following drastically different philosophies. More specifically, one would expect UF to perform much worse than TF with regard to *OMR* – one would almost expect UF to exhibit overall miss ratios at or around 1.0. The reader, however, is reminded that we are dealing with the UPUM case, where there is only one update process, i.e., a single update is active at any given time. Thus, when the update process goes to disk, the transaction processes are free to access the CPU. This explains why the UF process can process transactions even while giving the update process high priority. In the highly resource constrained MTUM case, discussed later in section 7.2.1, UF shows, as expected, *OMR*s of 1.0, as each update is regarded as a different process.

Having explored the real-time response of our system (i.e., miss ratios), we next turn our attention to how well our algorithms maintain state recency. This is measured through the *FSR* statistic, which measures what fraction of the reads inside the system accessed stale values. This metric is plotted at various system loads in figure 3D. The UF curve for *FSR* remains fairly constant, as would be expected by UF's policy of awarding higher priority to updates. The flatness of the curve indicates that UF is successful at retaining a moderate level of recency at all system loads. However, even at low transaction loads, *FSR* is not zero because of the unfairness of the FIFO queueing policy of updates. While UF would clearly install a large fraction of updates (i.e., missing very few), due to the unpredictability of update arrival patterns, it so happens that it may ignore some small set of data items for extended periods of time. This leads to the fairly high *FSR* even at low arrival rates. VDU, on the other hand, fueled by its fair update installation policy, has zero *FSR* at very low arrival rates and does substantially better than the other two at low transaction loads. However, its performance degrades at higher loads and stabilizes around loads of 30 transactions/sec. However, even at highly loaded conditions, it exhibits the lowest *FSR* values.

So far we were considering cases where stale reads were undesirable, but were not considered fatal. Now we turn our attention to a scenario where recency violations are not permissible, i.e., stale reads result in transaction aborts. Such requirements are typical of environments that require a high degree of precision, e.g., military command and control scenarios, where reading of stale data may result in missiles missing targets. To analyze this scenario, we are interested in seeing how many transactions make it without reading any stale values. This is expressed through the metric *SSV*, which is plotted in figure 3E. This figure is a strong testimonial to VDU's effectiveness. While UF and TF can hardly complete any transactions without stale reads, VDU completes a majority of transactions without stale reads at low loads and while its performance degrades at high loads, it still remains much superior to the performance of the other two. Basically, the reason for VDU's good performance is the combination of efficiency and fairness in update installation and judicious priority assignment in time cognizant scheduling.

7.1.2 Reduced Resource Contention

So far we were dealing with a uniprocessor system, where both updates and transactions were competing for very restricted resources. Now we explore the effect of reducing resource contention by setting *NumCPU* to 2 and *NumDisk* to 4. We plot the same metrics as before in figure 4. The trends of these curves are basically the same as those in the previous section. Two items of importance emerge: (a) Because of decreased resource contention, overall system performance improves across the board in the multiprocessor case; and (b) Most importantly, the performance of VDU improves more than the other two, indicating VDU is able to take advantage of the increased resources better than UF and TF. The second point is particularly manifest in figure 4E, where UF and TF show significantly worse performance than VDU compared to the uniprocessor case. In fact at low arrival rates, VDU is able to complete almost all transactions without stale reads. We take this to be an indication of the degree of fairness we are able to maintain in our update installation.

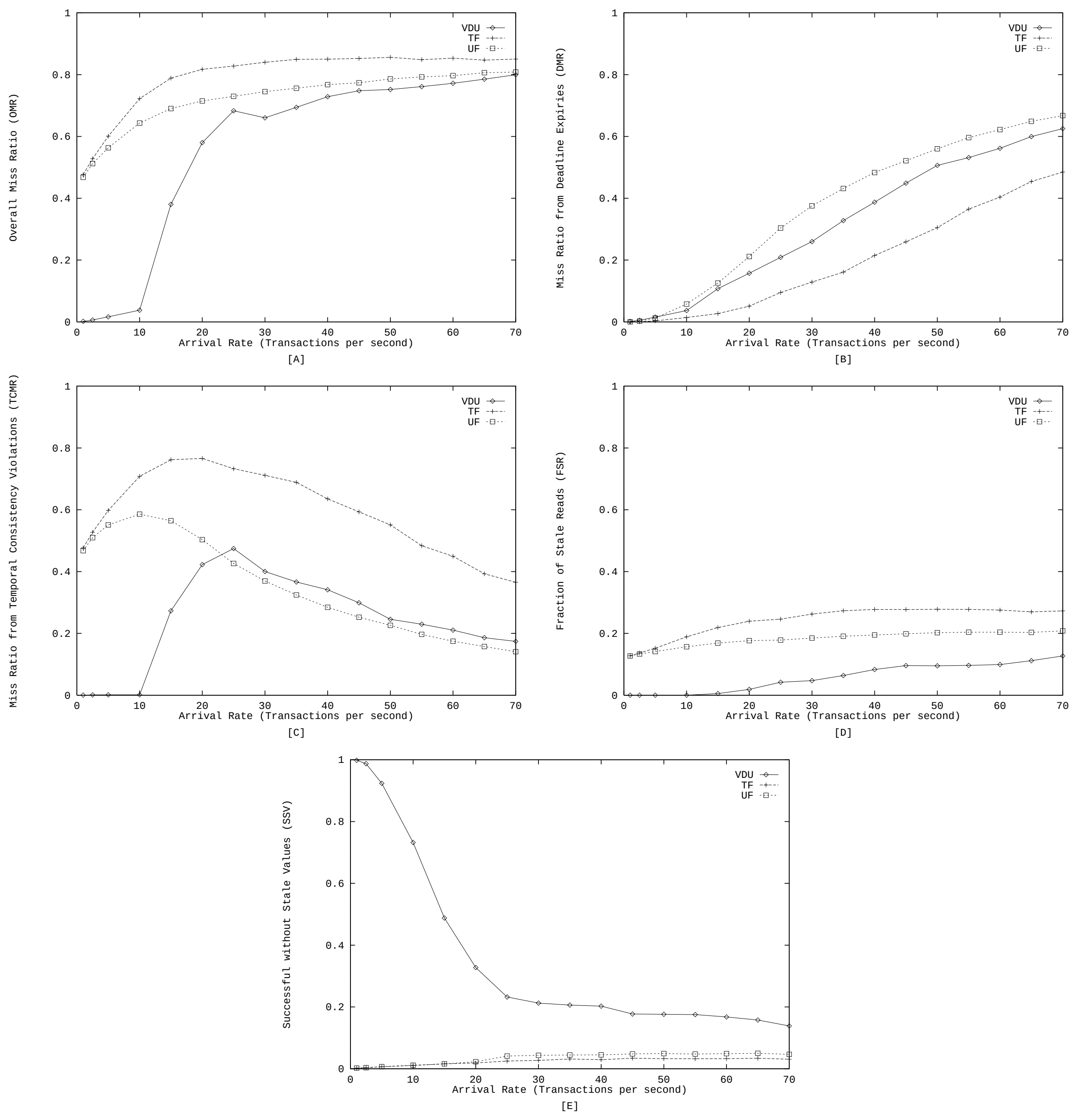

Figure 4: Experiments under UPUM, reduced resource contention

7.2 MTUM

Next we consider the MTUM update model.

7.2.1 High Resource Contention

Again, as in the UPUM case, we consider a uniprocessor system with only 2 disks to enforce high resource contention levels. Figure 5 plots the metrics.

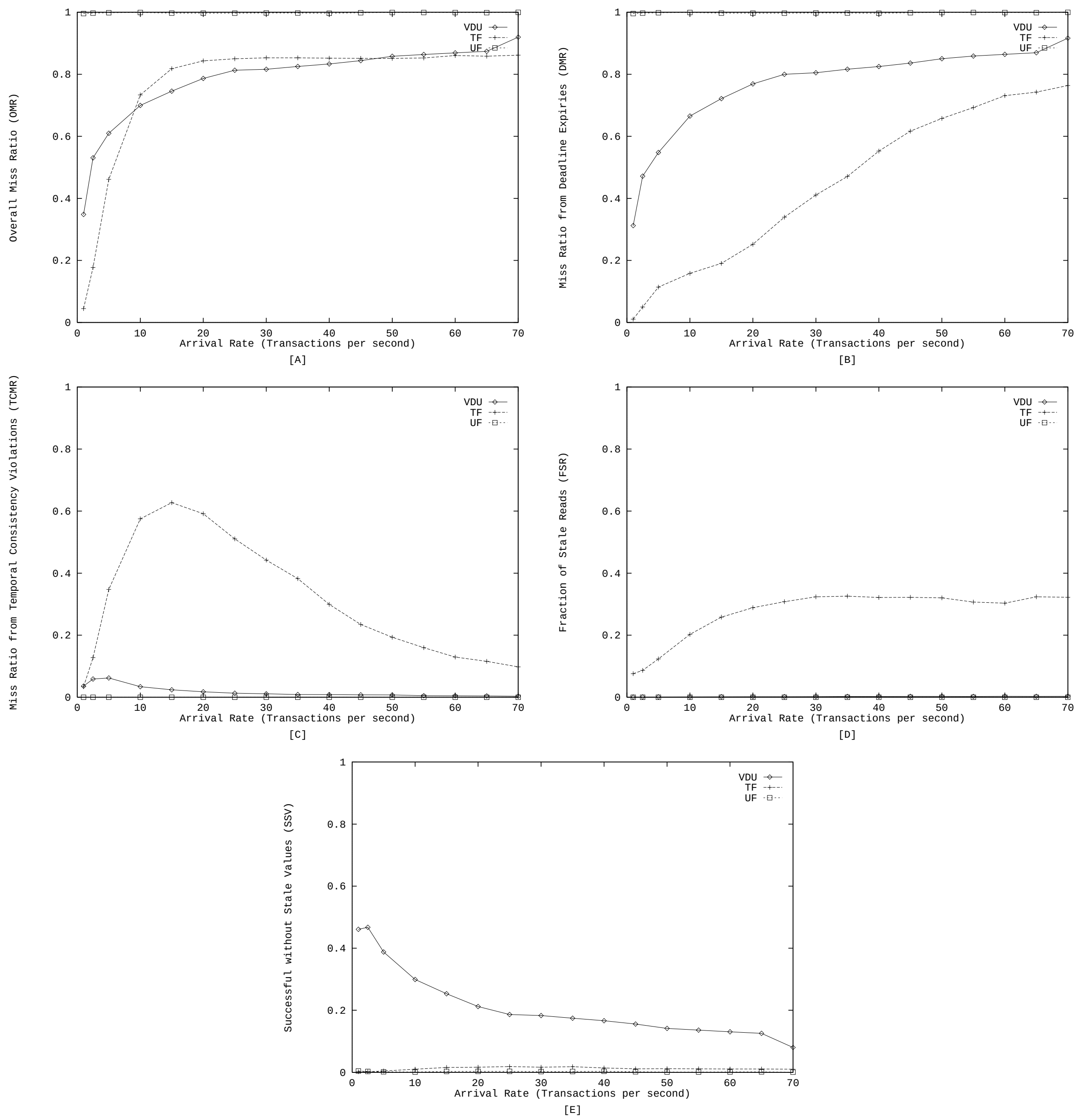

Figure 5: Experiments under MTUM, high resource contention

Figure 5 displays certain differences between the UPUM and the MTUM cases. First of all, UF shows a very high *OMR* (1.0) consistently. This is because of the fact that now, transactions have to compete with many updates, rather than a single update process as in UPUM. This results in nearly every transaction missing its deadline, leading to the inordinately high *DMR* and *OMR*. Another noteworthy feature of these experiments is that VDU is no longer the clear cut winner in terms of *OMR*. At high arrival rates, TF performs better in terms of the overall miss ratio than VDU. The reason for high *OMRs* at heavy loads is the same as that for UF. Because of the large number of update transactions, and also because individual update transactions now incur costs that they did not in the UPUM case (context switch), lesser numbers of transactions are able to complete. TF however, by assigning unilaterally high priorities to transactions, is immune from the effect of many update transactions. However, TF pays for its better *OMRs* by having substantially higher FSRs than VDU, as a result of discriminating against updates. Remarkably enough, while VDU does exhibit higher miss ratios than TF at heavy system loads, it is able to maintain a very low FSR, practically at zero. As a result, VDU outperforms TF substantially in terms of SSV.

7.2.2 Reduced Resource Contention

Results are reported in figure 6. In this case, the curves are as expected. VDU again regains its superiority, by virtue of being able to allocate resources more easily to transactions. By the same token the performance of UF also improves in comparison to the uniprocessor case. One remarkable feature of the *OMR* curves in figure 6 is the absence of the trough that was so noticeable in *OMR* curves so far. Recall that this trough was a result of the *TCMR* curve starting its downward trend, while still dominating the *DMR* curve. In this case however, the *TCMR* values are much lower compared to the other cases (e.g., the highest TCMR value is 0.58, while values over 0.8 were observed before). As a result *TCMR*s never really dominate *DMR*s. Thus the *OMR* curves are heavily influenced by the *DMR* curves and consequently do not display the dip.

7.3 Discussion

In this section we briefly attempt to generalize our findings:

1. VDU appears to be the best performer, chiefly because of its efficient and fair update installation policy and judicious priority assignment mechanism. More than efficiency (which is undoubtedly important), fairness seems to be a very important component in designing transaction processing systems for rapidly changing environments. In fact, for systems that cannot tolerate recency violations, fairness assumes an even greater importance. This is evidenced by VDU's clear superiority with regard to the *SSV* metric by virtue of its fair update installation procedure. In systems that can withstand some recency violations, a case may be made to award transactions higher priority than updates at high system loads and very tight resource contention levels. This is evidenced by TF exhibiting better *OMR* values than VDU in the uniprocessor MTUM case. Also, UF appears to be the worst performer overall, leading us to believe that it is probably a valid rule of thumb to bias the priority assignment process somewhat favorably towards transactions. This bias could take many forms, such as having more stringent tests to prioritize updates over transactions or reacting more strongly to violations of threshold MRs rather than threshold FSRs. We plan to study these and other changes to VDU in an extension to this paper.

2. There does not seem to be a significant difference in overall performance with regard to modeling updates as UPUM or MTUM. This is especially true at high system loads, where all the metrics seem to correspond fairly well under the two process models. This is a significant observation. In [3], the authors convincingly argue that UPUM is better. However, this does not seem to hold for disk resident systems, where disk fetches introduce delays. At these times, the added power of concurrency seems to negate the advantages reaped by having a single update process. Surprisingly, at

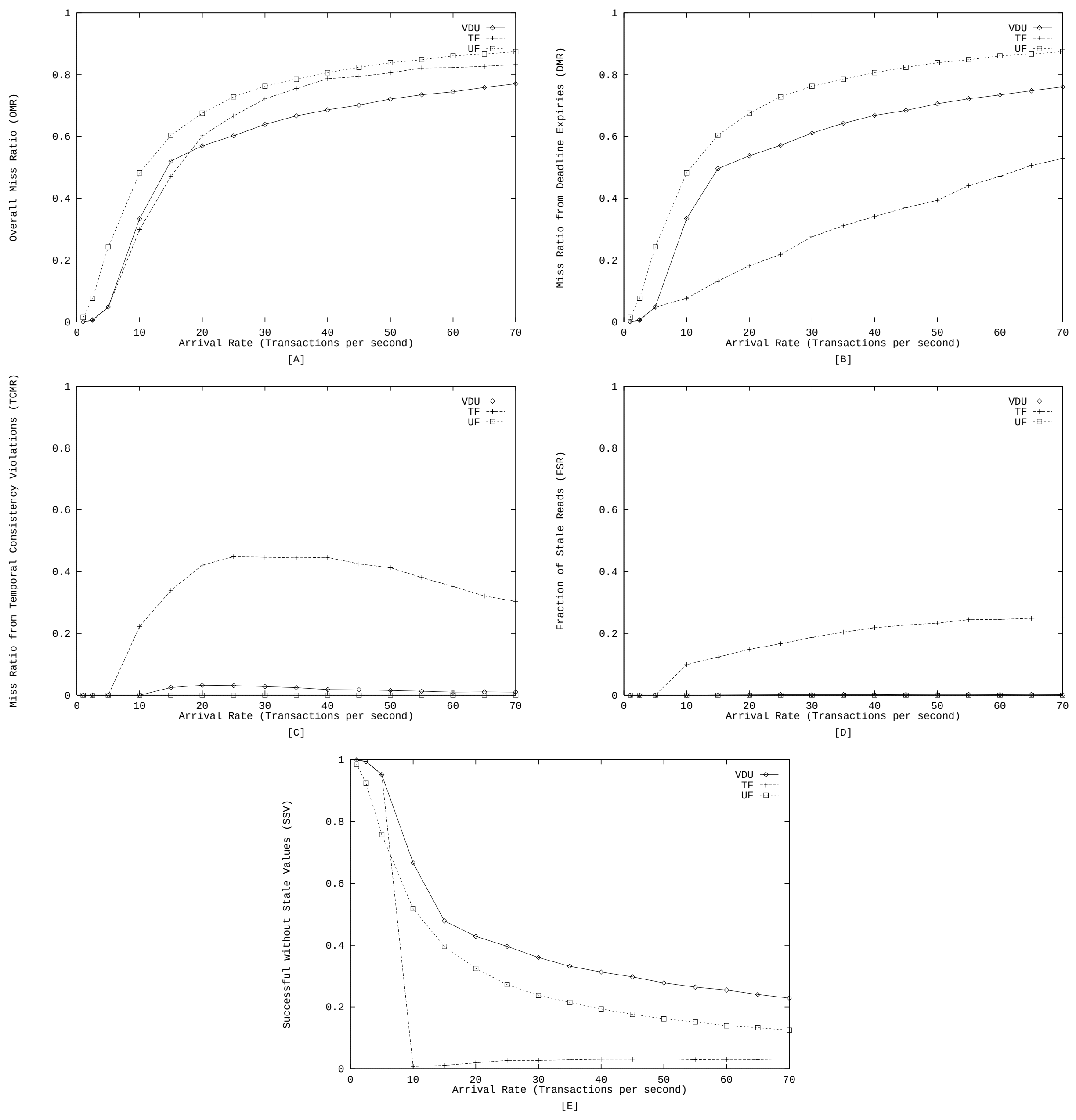

Figure 6: Experiments under MTUM, reduced resource contention

low system loads, MTUM seems to provide better performance particularly with regard to UF and TF. We are somewhat unwilling to axiomatize this finding, as we did not experiment how sensitive these findings are to changes in costs, such as context switch costs. In fact we plan to study this in an extension to this paper.

3. Finally a note on overhead. VDU's superiority does come at a price. There are two primary components to this overhead: (a) the cost of queueing updates in order of age of target data item,

and (b) adjusting feedback parameters α and β. Component (b) is not substantial, as the adjustment procedure is simple (trivial arithmetical operations) and it happens every *SampleBatch* commits, i.e, not very frequently. Component (a) is the major cost. It basically has two parts – a lookup of the *TS_ARRAY* and a prioritized queue insertion. The lookup can be done, with appropriate structures, in minimal constant time, while the insertion in the prioritized update queue is an $O(\log(N))$ operation. This does not appear to be too prohibitive a price to pay for the performance improvements the VDU policy provides.

8 Conclusion

In this paper we described our preliminary ideas on designing transaction processing systems for *rapidly changing systems*, an emerging class of systems. these applications require the amalgamation of *temporal* and *real-time* characteristics and consequently require transaction processing strategies that are able to balance both these facets. In this paper we described an integrated set of strategies to achieve the above – these consisted of an update installation as well as a scheduling policy. In the process of describing these algorithms, it was our intent to touch upon major issues that need to be considered. Such issues are efficiency and fairness of update installation and the need to balance update and transaction loads.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th VLDB*, 1989.

[2] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: Performance Evaluation. *ACM Transactions on Database Systems*, 1992.

[3] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying Update Streams in a Soft Real-Time Database System. In *Proceedings of the 1995 ACM SIGMOD*, pages 245–256, 1995.

[4] I. Ahn. Database Issues in Telecommunications Network Management. In *ACM SIGMOD*, pages 37–43, May 1994.

[5] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, 1986.

[6] A. Bestavros and S. Braoudakis. SCC-nS: A family of Speculative Concurrency Control Algorithms for Real-Time Databases. In *Proceedings of the 3rd Intl. Workshop on Responsive Computer Systems*, 1993.

[7] R. L. Blum and G. Wiederhold. Studying hypotheses on a time-oriented clinical database: An overview of the rx project. In *Proceedings of the Symposium on Computer Applications in Medical Care*, pages 725–735, Washington, DC, 1982. IEEE Computer Society.

[8] A.P. Buchmann and H. Branding. On Combining Temporal and Real-Time Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages F1–F5, 1993.

[9] Sprint Network Management Center. Site Visit, April 1992.

[10] S. Chakravarthy and S-K. Kim. Semantics of Time Varying Information and Resolution of Time Concepts in Temporal Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages G1–G13, 1993.

[11] J. Clifford and A. Tansel. On An Algebra for Historical Relational Databases. In *Proceedings of the International Conference on Management of Data* , pages 247–265, 1985.

[12] A. Datta. Research Issues in Databases for Active Rapidly Changing data Systems (ARCS). *ACM SIGMOD RECORD*, 23(3):8–13, September 1994.

[13] A. Datta. Databases for Active Raidly Changing data Systems (ARCS): Augmenting Real-Time databases With Temporal and Active Characteristics. In *Proceedings of the First International Workshop on Real-Time Databases*, pages 8–14, March 1996.

[14] A. Datta, S. Mukherjee, P. Konana, I. Viguier, and A. Bajaj. Multiclass Transaction Scheduling and Overload Management in Firm Real-Time Database Systems. *Information Systems*, 21(1):29–54, March 1996.

[15] O. Etzion, A. Gal, and A. Segev. Temporal Active Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, June 1993.

[16] P.A. Fishwick. Simpack: Getting started with simulation programming in C and C++. Technical Report TR92-022, Computer and Information Sciences, University Of Florida, Gainesville, Florida, 1992.

[17] P.A. Fishwick. *Simulation Model Design And Execution: Building Digital Worlds.* Prentice Hall, 1995.

[18] S.K. Gadia. A homogeneous relational model and query languages for temporal databases. *tods*, 13(4):418–448, dec 1988.

[19] S.K. Gadia and G. Bhargava. Sql-like seamless query of temporal data. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, jun 1993.

[20] S.K. Gadia and J.H. Vaishnav. A query language for a homogeneous temporal database. In *pods*, pages 51–56, mar 1985.

[21] J. Haritsa, M. Livny, and M. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1991.

[22] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. *ACM PODS*, 1990.

[23] J.R. Haritsa, M.O. Ball, N. Roussopoulos, A. Datta, and J. Baras. MANDATE: MAnaging Networks using DAtabase TEchnology. *IEEE Journal on Selected Areas in Communications*, 11(9):1–13, 1993.

[24] J.R. Haritsa, M.J. Carey, and M. Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1990.

[25] J.R. Haritsa, M.J. Carey, and M. Livny. Data Access Scheduling in Firm Real-Time Database Systems. *The Journal of Real-Time Systems*, 4, 1992.

[26] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *Proceedings of the 17th Intl. Conf. on Very Large Data Bases*, 1991.

[27] J. Huang, J. Stankovic, D. Towsley, and K. Ramamrithnam. Experimental Evaluation of Realtime transaction processing. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[28] C. Jensen, J. Clifford, S.K. Gadia, A. Segev, and R.T. Snodgrass. A Glossary of temporal Database Concepts. *ACM SIGMOD RECORD*, 21(3):35–43, 1992.

[29] C.S. Jensen and R. Snodgrass. Semantics of Time-Varying Attributes and Their Use for Temporal Database Design. To appear in *Proceedings of the Object-Oriented and Entity Relationship Conference*, Gold Coast, Australia, December, 1995. Also Technical Report R-95–2012, Aalborg University, Department of Mathematics and Computer Science, Frederik Bajers Vej 7E, DK–9220 Aalborg Øst, Denmark, May 1995.

[30] C.S. Jensen and R. Snodgrass. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):954–974, 1994.

[31] G. Koren and D. Shasha. D^{over}: An optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 290–299, August 1992.

[32] A.M. Law and W. Kelton. *Simulation Modeling and Analysis.* McGraw Hill, 1991.

[33] J. Lee and S.H. Son. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1993.

[34] T.Y.C. Leung and R. Muntz. *Stream Processing: Temporal Query Processing and Optimization*, chapter 14, pages 329–355. Benjamin/Cummings, 1993.

[35] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, January 1973.

[36] D. Menasce and T. Nakanishi. Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management. *Information Systems*, 7(1), 1982.

[37] H. Pang, M. Livny, and M.J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[38] K. Ramamritham. Real-Time Databases. *Distributed and Parallel Databases: An International Journal*, 1(2):199–226, 1993.

[39] K. Ramamritham. Time for Real-Time Temporal Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages DD1–DD5, 1993.

[40] K. Ramamritham, R. Sivasankaran, J.A. Stankovic, D.T. Towsley, and M. Xiong. Integrating Temporal, Real-Time and Active Databases. *ACM SIGMOD RECORD*, 25(1):8–12, March 1996.

[41] B. Salzberg and V. Tsotras. A Comparison of Access methods for Time Evolving Data. Technical Report NU-CCS-94-21, College of Computer Science, Northeastern University, 1994.

[42] A. Segev and H. Gunadhi. Query Processing in Temporal Databases. In *Proceedings of the Workshop on Query Optimization*, pages 159–164, Portland, Oregon, may 1989.

[43] R. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.

[44] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of ACM SIGMOD*, June 1985.

[45] X. Song and J.W.S. Liu. How Well Can Data Temporal Consistency be Maintained? In *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, 1992.

[46] M.D. Soo. Bibliography on Temporal Databases. *ACM SIGMOD RECORD*, 20(1):14–23, 1991.

[47] R.M. Verma and P.J. Varman. Efficient Archivable Time Index: A Dynamic Indexing Scheme for Temporal Data. In *Proceedings of the International Conference on Computer Systems and Education*, pages 59–72, 1994.

[48] G. Wiederhold. How to write a schema for a time oriented medical record data bank. Technical report, Stanford, 1973.

[49] G.T.J. Wuu and U. Dayal. A Uniform Model for Temporal Object-Oriented Databases. In *Proceedings of the International Conference on Data Engineering*, pages 584–593, 1992.