Real-Time Logging and Failure Recovery

LihChyun Shu
Dept. of Information Management
Chang Jung University
Tainan County, Taiwan 711, ROC
shulc@mail.cju.edu.tw

John A. Stankovic and Sang H. Son
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
{stankovic,son}@cs.virginia.edu

21 November 2001

Abstract

Real-time databases are increasingly being used as an integral part of many computer systems. During normal operation, transactions in real-time databases must execute in such a way that transaction timing and data time validity constraints can be met. Real-time databases must also prepare for possible failures and provide fault tolerance capability. Principles for fault tolerance in real-time databases must take timing requirements into consideration and are distinct from those for conventional databases. We discuss these issues in this paper and describe a logging and recovery technique that is time-cognizant and is suitable for an important class of real-time database applications. The technique minimizes normal runtime overhead caused by logging and has a predictable impact on transaction timing constraints. Upon a failure, the system can recover critical data to a consistent and temporally valid state within predictable time bounds. The system can then resume its major functioning, while non-critical data is being recovered in the background. As a result, the recovery time is bounded and shortened. Note that the results presented in this paper depend on pre-declared and periodic critical transactions and non-volatile RAM for critical data logging. Our performance evaluation via simulation shows that logging overhead has a small effect on missing transaction deadlines while adding recovery capability. Experiments also show that recovery using our approach is 3 to 6 times faster than traditional recovery.

1 Introduction

In recent years, with the advances in hardware and networking technologies, more and more real-time database (RTDB) applications are emerging. Many of these applications, such as air traffic control, network management, internet programmed trading, and command and control systems demand predictable low-latency access to data, coupled with stringent durability and availability requirements. Data in such applications often has distinguishing characteristics [SSH99]. For instance, some data has temporal validity intervals associated with them. During a validity interval, a particular data value is deemed useful as far as the application's semantics is concerned. Data gets out of date by the simple passage of time. On the other hand, some real-time data is more critical to the operation of a real-time application than other data. As an example, consider an internet programmed trading application where transactions are submitted from around the world 24 hours a day. In such a system, customer balances constitute critical data and the state of each customer's balance will remain valid until a transaction is issued to change it. This data does not become invalid simply with the passage of time. On the other hand, stock market prices are also critical, but have finite validity intervals; a stock price that is too old is worthless.

This work was supported, in part, by NSF grant EIA-9900895 and by NSC grant 88-2213-E-309-002.

Following the terminology used by Ramamritham in [Ram93], we call those data whose values will not change with time or whose validity intervals are infinite *invariant* data. Other data is called *variant* data.

When a failure occurs in these applications, it is important that the system can recover critical data to a consistent and temporally valid state within predictable time bounds. The system can then resume its major functioning, while non-critical data is being recovered in the background. Principles that are commonly addressed in conventional database failure recovery do not take real-time requirements into consideration. In this paper, we first define principles which are appropriate for logging and recovery in RTDB applications. We then present a logging and recovery technique that is time-cognizant, supports these principles and, is suitable for a class of real-time database applications, such as internet programmed trading. The key features of our scheme are as follows:

- We allow a system designer to specify which data is critical to the system's major operation. We assume transactions that update critical data are also critical. Critical transactions are assumed to update only critical data.

- We partition the log across critical and non-critical data segments. Partitioning the log this way allows critical data to be recovered independently. Transactions that only access critical data can start executing before non-critical data has been recovered.

- In order to reduce and bound logging overheads and post-crash recovery time, we store critical data in non-volatile RAM.

- We employ different logging strategies for different data types. For each critical variant datum, we associate with each of its states a last valid time instant, which will be used to determine whether the corresponding state is still valid when the system restarts after a failure. We assume real-time data updated by each transaction becomes temporally valid when the updating transaction commits. Thus, we can assign a single last valid time instant to all after images of critical variant data updated by a transaction when the transaction commits. The write ahead logs for such data contain only the redo records of committed transactions; this minimizes recovery I/O. On the other hand, the write ahead log for critical invariant data contains only the undo records of active transactions. When the updating transaction commits, after-images of such data are flushed, and the undo log for the committed transaction are discarded. The write ahead log for non-critical data contains only the undo records of active transactions. The impact of our logging schemes on the timeliness of critical transactions is amenable to pre-run-time timing analysis since the critical transactions are pre-declared.

- The log records for critical variant data are organized according to last valid time instant of the datum stored in each log record. This structure permits us to quickly determine which critical data can be restored from the persistent log (because this data is still temporally valid when the system resumes its operation) and which critical data must be refreshed by appropriate transactions and/or sensor readings. We assume critical transactions that update variant data are periodic. Based on this assumption, we can calculate the worst-case number of transactions which have produced log records containing valid data. As a result, the recovery time for critical variant data is predictable.

- The recovery procedure after a system crash makes only a single pass over each log, because we separate the undo logs from the redo logs.

- Application properties are exploited to design and improve the performance of our recovery algorithm. For example, deallocation of the storage space for log records containing invalid data is made easier, because we organize these log records according to last valid time instant of the datum stored in each log record.

We organize this paper as follows. Section 2 discusses related work. Section 3 describes basic concepts and introduces some terms used in the remainder of the paper. Section 4 discusses principles that are appropriate for logging and recovery in RTDB applications. Section 5 details our logging and failure recovery algorithm, its characteristics, and advantages. Section 6 shows the performance evaluation of our approach for various system parameters. Section 7 concludes the paper.

2 Related Work

Real-time logging and failure recovery addresses the problem of restoring a RTDBS upon a memory failure or crash, using logs created during normal operation, to a consistent state so that a recovery time requirement can be met. During normal runtime, one also wants to minimize runtime effort due to logging and have a predictable impact on transaction timing constraints. Research in this domain distinguishes itself from that in traditional database recovery mainly in the following two requirements: first, upon a crash, it is important for a RTDBS to predictably come back within a pre-determined time bound and open up to the environment again; second, when the system resumes its operation, data in a RTDBS must be temporally valid or be made so. Hence, while some data may be recovered from existing data in the logs, other data must be refreshed by reading environment states, e.g., reading the current stock prices. When these requirements are satisfied, one can then be dealing with controlling and/or monitoring real-time environments.

Considerable research has been done in traditional database recovery. Two recent books by Kumar and Hsu [KH98] and Kumar and Son [KS98] discuss recovery in detail and contain descriptions of recovery methods used in a number of existing relational database products. Research in traditional database recovery generally places emphasis on performance issues. Timing predictability is seldom a concern in this field. As an example, consider the ARIES recovery algorithm [MHL+92] which has been quite successful in practice. ARIES uses a steal/no-force approach for writing, and it is mainly based on three concepts: (1) write-ahead logging, (2) repeating history during redo, and (3) logging changes during undo. Write-ahead logging ensures that consistent database state can be recovered at recovery time. Repeating history during redo means that transaction execution history prior to crash will be retraced and reapplied to reconstruct the database state. The REDO phase is preceded by an analysis step, which determines the start point for REDO. In addition, information stored by ARIES and in the data pages allows ARIES to apply only the necessary REDO operations during recovery. The third concept, logging during undo, will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process. While ARIES has employed several novel techniques to optimize its recovery-time performance, it cannot ensure that restored data is temporally valid, nor can it ensure that the system will come back on line in time. In short, what ARIES lacks for a RTDB is time-cognizant protocols for logging and recovery.

Compared to research done in traditional database recovery, relatively little research has been done for recovery in RTDB. Sivasankaran et al.[SRST95] look at the characteristics of data that are present in real-time active database systems and discuss how to do data placement, logging and recovery to meet the performance requirements. They also discuss transaction characteristics that can influence the data placement, logging and recovery in real-time active databases. Sivasankaran et al.[SRS97] show the need to design novel logging and recovery algorithms by observing the "priority diversion" problem[1] where conventional logging and recovery algorithms are not suitable in a priority oriented RTDB setting. They present a taxonomy of data characteristics and propose two data classes that are derived from data types and transaction types. They also develop a suite of algorithms targeted at RTDB. The major differences between our work presented in this paper and that presented in [SRS97] are the following: first, we propose principles underlying real-time logging and failure recovery, independent of technologies used; second, one of our major concerns for logging and recovery is maintaining temporal consistency of data; third, we employ different logging strategies for different data types; fourth, we exploit application properties to improve the performance of our recovery algorithm; finally, we clearly characterize the impact of logging, commit processing, and recovery on satisfying transaction timing constraints and post-crash performance requirements.

RTDB recovery mechanisms based on the shadowing technique, instead of logging, were investigated by Shu et al. [SSK99]. One of the advantages of using shadowing-based recovery technique, compared to logging, is reduced post-crash recovery time (requires neither redo nor undo). However, some shadowing techniques can incur significant overheads at normal run-time, which must be minimized and bounded for real-time applications.

3 Background and Assumptions

Real-time systems (RTS) must react to stimuli from the environment within time intervals dictated by the environment. Hence, the state of the operating environment is constantly monitored by a RTS. We assume the database $D = \{x_1, x_2, \ldots, x_n\}$ and a subset of the data items in D be a representation of the operating environment. Suppose $x_i, 1 \le i \le l$, represents x_i^e in the external environment. We term each x_i an *internal* variable and x_i^e an *external* variable. We let $D^E = \{x_1, x_2, ..., x_l\}$ and $E = \{x_1^e, x_2^e, ..., x_l^e\}$.

The state of the operating environment continuously changes; therefore, at any instant, the external variables reflect changes about the operating environment they represent. The changes in the operating environment are mapped to D^E with appropriate modifications to D^E. If $S_1, S_2, ..., S_k$ are a sequence of state changes for D^E, then there exists a sequence of state changes for E, $S_1', S_2', ..., S_k'$, such that $S_i(x_j) = S_i'(x_j^e)$ for $1 \le i \le k, 1 \le j \le l$.

The state of the operating environment changes mainly due to two types of actions, namely *environment-activated* or *system/user-activated*. For example, air temperature change is typically caused by environment-activated action, e.g., the flow of air. On the other hand, a customer balance changes its state by the actions of user-activated financial transactions. A product manufactured on an assembly line changes its state by the actions of system-controlled robot arms. Because the external environment changes state from time to time, we can expect that a particular state may remain valid for a limited period of time. The *temporal validity interval* of a state of an external variable is defined as the time span from the time the state is generated (by either environment- or system/user-activated action) until the time the external variable changes to a new state. We define the temporal validity interval of an external variable x_j^e, denoted as $TVI(x_j^e)$, as the minimum of the temporal validity intervals of all possible states of x_j^e. The temporal validity interval of a state of an internal variable is defined as the time span from the time the state is generated in the system until the time the state is no longer usable as far as the application's semantics is concerned. We define the temporal validity interval of an internal variable x_j, denoted as $TVI(x_j)$, as the minimum of the temporal validity intervals of all possible

[1] Priority diversion occurs when high priority requests do work for low priority requests, e.g., committing a transaction T by flushing log records belonging to both T and other lower priority transactions.

states of x_j. Observe that if an internal variable x_j changes its state due to system/user-activated actions, then it is often the case that any state of such a variable will not change with time unless a new action is initiated to change the variable to a new state. Such data is termed *invariant* data while other data is termed *variant* data.

We define $LVI(T_i, x_j, H)$ to be the last time instant when the value of x_j written by T_i remains valid where H is a history that includes $W_i[x_j]$.

The temporal validity interval is a distinguishing attribute of real-time data. Real-time data has other attributes. For example, some data in D are more critical for the normal operation of real-time systems than others. We denote critical data in D as $D_{critical}$. Transactions that access real-time data can also be classified into different types. For a detailed taxonomy of attributes associated with real-time data and transactions, interested readers are referred to [SRS97].

In this paper, we assume transactions pre-declare their data needs. We assume transactions that update critical data are also critical. Critical transactions are assumed to update only critical data. Further, we assume the number of critical transactions is known at design time. Critical transactions that update variant data are periodic. Consider the internet trading example again. Stock prices are critical variant data. They are refreshed periodically by critical sensor transactions. Customer balances are critical invariant data and are updated by on-demand trading transactions. Other data such as the number of transactions being processed today is non-critical data.

4 Principles Underlying Real-Time Logging and Failure Recovery

Conventional database failure recovery is primarily concerned with keeping the database in a consistent state. Principles such as atomicity and durability commonly addressed in database failure recovery do not take real-time requirements into consideration. In this section, we describe principles which are appropriate for fault tolerance in real-time database applications. In Section 5, we will present a logging and recovery technique that is time-cognizant, supports these principles, and is suitable for a class of real-time database applications, such as internet programmed trading. In an internet programmed trading application, transactions may be submitted to a web server from around the world 24 hours a day. The success of such an application might be significantly impaired if, upon a failure, the system can be recovered sometimes within a few minutes, but sometimes it takes several hours. Maintaining a predictably high level of post-crash recovery performance not only reduces direct costs of downtime, but also helps the business gain other intangible benefits, e.g., higher customer loyalty. As real-time databases are increasingly being used as an integral part of many computer systems, predictably fast recovery upon a failure may soon be an important enabling technology. It may also become a desirable technology for some of today's and tomorrow's global e-business that requires consistent 24*7 levels of service.

To maintain the consistency of data despite system failures, real-time databases must perform book-keeping activities, e.g., logging, during the normal operation of the system. Logging writes on stable storage all updates done to the database. These activities imply extra run-time overheads, which must be amenable to pre-run-time timing analysis for real-time databases. In particular, one must bound and minimize the size of each log. Further, the overheads in maintaining each log structure must be predictably accounted for. Typically, these involve both memory and I/O operations. In a nutshell, logging activities must not jeopardize the timeliness of transactions.

Most real-time data has limited temporal validity intervals. As a consequence, when a system failure occurs, we should only restore persistent data which will remain temporally valid when the system resumes execution. As in conventional database recovery, the restored value of each data item x_j must be created by the last committed writer of x_j prior to the crash in the execution history. To determine whether x_j's restored value is temporally valid, we make use of x_j's last valid time instant. However, testing temporal validity for each individual datum can be expensive. In Section 5, we present a technique to test temporal validity for a group of data items with close last valid time instants. If x_j's restored value is deemed to be temporally invalid, then the value of x_j will need to be refreshed by a new transaction when the system restarts. Note that if the last committed state of any datum is not temporally valid, then no other committed state of the same datum will.

Some real-time database applications not only require fast recovery upon a failure, but also predictably fast recovery. It is important to recognize that several time segments must be taken into consideration in the design of database recovery, since a couple of key events must happen before we can actually recover the database. They include the system downtime, the time to reboot the operating system, and the time to reboot the database system. We assume all these times can be made predictably short[2].

Apropos recovering the database itself, it is often the case that it is not necessary to recover the entire database before the system can resume its major functioning. Instead, we can first recover data in $D_{critical}$. In the next section, we present a recovery technique that can recover critical data within predictable time bounds. Non-critical data can be recovered when the system has spare capacity.

5 A Real-Time Logging and Recovery Algorithm

In this section, we present a logging and failure recovery algorithm for RTDB. In order to make data write operations for critical transactions predictable, we place critical data in main memory. We assume the availability of non-volatile RAM (NVRAM, also called solid state disk) [CKKS89] for stable storage. NVRAM has two important characteristics: its access speed is comparable to volatile RAM and it is as stable as a disk. In terms of the access performance, conventional disks can do about 125 I/Os per second, NVRAM can do about 6000 I/Os per second [Jor]. Although we cannot afford to have the entire database in NVRAM, we can keep a backup copy of critical invariant data in NVRAM in order to support the principle of durability for temporally valid data. Such data are forced at transaction commit time in our logging schemes described below. Using NVRAM, instead of disks, to store such data can reduce commit time and contribute to predictable transaction execution times. Critical variant data is not archived in stable storage. At recovery time, such data is recovered solely from the log records.

Operations on log records for critical data must also be predictable. In Section 5.1, we will describe our organization of log records for critical data as multiple logs. In order for both execution time and recovery time for critical transactions to be predictable, we place these logs in NVRAM. For non-critical data, a single log is maintained and it resides on a separate disk drive. We summarize the data placement policy for our real-time logging and recovery algorithm in Table 1.

Conflict resolution for concurrency control is *priority abort*, where the conflicting transaction with lower priority waits or gets aborted depending on whether it is the requester or holder of locks, respectively.

5.1 Conditional Logging

Because real-time data have distinguishing characteristics, we can leverage on this information to design proper logging strategies for different data types. Three different types of write-ahead logs are maintained in our system:

[2] If the assumption is not true, e.g., say the processor is down for one hour and all data has validity intervals less than one hour, then the set of timed data that can be recovered is empty.

Table 1: Data placement policy and other key features for our real-time logging and recovery algorithm

data type	stable storage for data	stable storage for log	FORCE	STEAL	system-wide checkpoints
$D^{var}_{critical}$	—	NVRAM	NO	NO	NO
$D^{invar}_{critical}$	NVRAM	NVRAM	YES	NO	NO
$D_{non-critical}$	disk	disk	YES	NO	NO

* Critical data is placed in main memory. Critical invariant data is also backed up in NVRAM in order to support the durability and predictability principles. Critical variant data is backed up solely in log records stored in NVRAM.

one or more redo logs for $D^{var}_{critical}$ (critical variant data), a single undo log for $D^{invar}_{critical}$ (critical invariant data), and a single undo log for $D_{non-critical}$ (non-critical data).

We assume data items in $D_{critical}$ updated by each transaction become temporally valid when the updating transaction commits. As a result, data items in $D^{var}_{critical}$ updated by the same transaction have the same last valid time instant (LVI), i.e., they become temporally invalid at the same time. If a datum x is updated by both T_i and T_j and T_i commits before T_j, then we assume the time the value of x created by T_i becomes temporally invalid is no later than the time the value of x created by T_j becomes invalid. In other words, the LVI associated with the value of x created by T_i is no greater than the LVI associated with the value of x created by T_j.

5.1.1 Logging for critical variant data

For data in $D^{var}_{critical}$, the system maintains in memory a private redo log for each active transaction. The system also maintains one or more redo logs for data in $D^{var}_{critical}$ on NVRAM, denoted as $CriticalRedoLog_1$, $CriticalRedoLog_2, \ldots$. We refer to the log in stable storage as a *persistent* log and the log in memory as a *volatile* log. The redo records of an active transaction are kept initially in a private redo log in main memory, and these redo records are appended to a persistent redo log, based on a criterion described below, only when the transaction begins its commit processing[3]. If a transaction is aborted, then the volatile redo log of the transaction is simply discarded.

In order to do away with undo logging for data in $D^{var}_{critical}$, we use a deferred updates technique. In other words, updates done on data items in $D^{var}_{critical}$ by an uncommitted transaction are noted without executing them. The deferred updates are installed in main memory when the updating transaction begins its commit processing.

The reason we maintain one or more persistent redo logs for data in $D^{var}_{critical}$ is that at recovery time we can quickly identify which logs contain data that will remain valid when the system resumes its operation. To do so, we associate each $CriticalRedoLog_i, i \geq 1$, with two attributes $ELVI_i$ and $LLVI_i$, which record the earliest and latest "last valid time instant" of all log records currently stored on $CriticalRedoLog_i$, respectively. We make $LLVI_i < ELVI_{i+1}$, for every $i \geq 1$. The redo records of an active transaction kept in a private redo log are given a single *LVI* when the transaction begins its commit processing. These records are then appended to a persistent redo log $CriticalRedoLog_j$, such that $ELVI_j \leq LVI \leq LLVI_j$. We illustrate this log structure in Figure 1.

[3]Because each active transaction has a separate private redo log associated with it, priority diversion will not occur when these logs are flushed to stable storage.

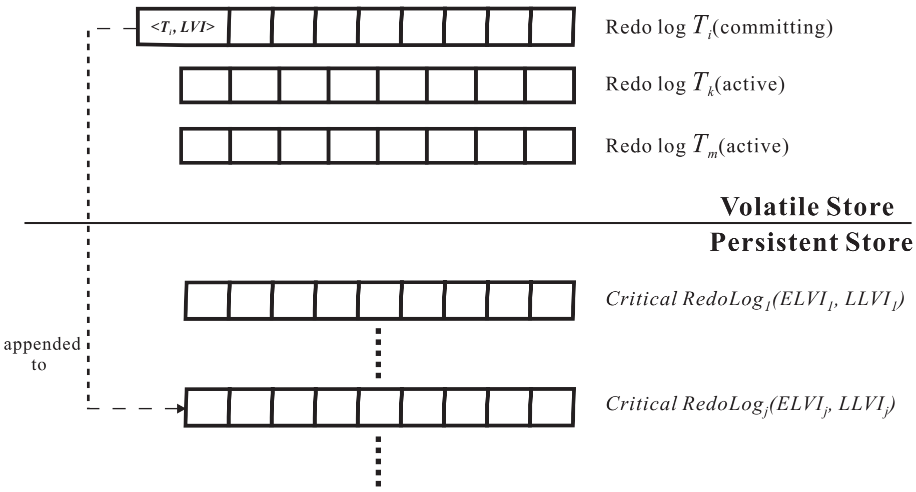

Figure 1: The log structure for critical variant data

In Section 5.3.1, we describe an iterative checking algorithm that uses these $ELVI$s to identify logs containing data that will remain valid when the system resumes its operation. The idea is to consider each log as an atomic unit, i.e., either all data in a redo log is treated as useful or none of them is deemed useful as far as temporal validity is concerned. Let $DLVI_i = LLVI_i - ELVI_i$ and $DLVI = max_{i\geq 1} DLVI_i$. Observe that if $DLVI$ is chosen to be large, then more valid data may be discarded when a redo log on which the data reside is determined not to be useful. On the other hand, if $DLVI$ is small, then fewer valid data may be discarded for each log deemed not useful. However, because there may be more logs to check when $DLVI$ is small, more time will be spent in the checking process, which in turn may affect the final outcome of the checking process in determining useful redo logs. In addition, with a small $DLVI$, more storage space will be needed for record keeping. In Section 5.3.3, we derive the bounds for the maximum number of log records that contain valid data in the redo logs and the maximum number of redo logs. Knowing these values permits us to bound the recovery time for critical data.

Another advantage we get from organizing redo records as described above is that garbage collection of the storage space for log records is made easier: If current_time $> ELVI_i$, then the storage space for $CriticalRedoLog_i$ can be reclaimed. Thus, provided we allocate sufficient space for these redo log records[4], checkpointing for critical variant data becomes unnecessary. One can design data structures for these logs so that storage allocation and reclamation can be made efficient. At recovery time, temporally valid critical data are restored from these log records. Other critical data are refreshed by appropriate transactions and/or sensor readings.

5.1.2 Logging for critical invariant data

For data in $D^{invar}_{critical}$, the system maintains in memory a private undo log for each active transaction T_i. Before T_i commits, T_i's private undo log for data in $D^{invar}_{critical}$ is flushed to become a persistent $CriticalUndoLog$. We then force data items in $D^{invar}_{critical}$ updated by T_i. The persistent $CriticalUndoLog$ can be discarded when T_i commits. The reason we force critical invariant data at commit is to eliminate checkpointing for such data. However, if such data is stored on disk, then forcing such data at commit implies negligible locality of data accesses in the disk,

[4]The amount of space needed for these redo logs can be pre-determined because we assume the following: (1) the number of critical transactions is known, (2) transactions pre-declare their data needs, and (3) each critical transaction executes with a pre-determined frequency.

which in turn can affect the timeliness of critical transactions. To cope with this problem, we use non-volatile RAM, instead of disk, as the stable storage for $D^{invar}_{critical}$. Because accessing non-volatile RAM is like accessing volatile RAM during normal run time, the impact of forcing critical invariant data at commit with respect to satisfying transaction timing constraints is predictable.

5.1.3 Logging for non-critical data

For data in $D_{non-critical}$, the system maintains a single undo log on the disk, denoted as *NonCriticalUndoLog*. Logging actions for non-critical data are similar to those for critical invariant data. One notable difference is that because non-critical data is not indispensable to resume a system's critical operation, we use the disk, instead of non-volatile RAM, as the stable storage for non-critical data.

5.1.4 Predictable impact of logging

Logging writes on stable storage all updates done to the database. These activities imply extra run-time overheads, which must be amenable to pre-run-time timing analysis for real-time databases. We summarize our logging schemes described above in Table 1.

In our logging schemes, for each updated datum, we need to create a log record (either redo or undo record), which is placed on an appropriate log. Because we assume log records are initially kept in volatile storage, maintaining redo and/or undo logs for a transaction involves memory write operations and additional overheads in linking together log records. Because we assume transactions pre-declare their data needs, we are able to calculate the total pre-commit overheads caused by logging. At the end of each committing transaction, we must write out private redo/undo records. This I/O overhead must also be accounted for. However, the overhead is kept small and predictable because we place persistent logs for critical data in NVRAM. Hence, transaction commit time is reduced. Note that our use of private logs reduces contention on the persistent log tails. The log tails are accessed only when a transaction is beginning to commit and repeated acquisition of short-term locks on the log tails is eliminated.

5.2 Commit Processing

Suppose T_i is a critical transaction. We denote data $\in D^{var}_{critical}$ updated by T_i as $D^{var}_{critical,i}$ and data $\in D^{invar}_{critical}$ updated by T_i as $D^{invar}_{critical,i}$. When T_i starts its execution, it is added to the list of active transactions. When T_i finishes executing, it pre-commits, which involves the following steps:

Pre-commit Processing:

- Install T_i's updates to data $\in D^{var}_{critical}$ in main memory.
- Determine the *LVI* for $D^{var}_{critical,i}$. The record $< T_i, LVI >$ is added to the private redo log for $D^{var}_{critical,i}$, and the private redo log is appended to a persistent $CriticalRedoLog_j$, such that $ELVI_j \leq LVI \leq LLVI_j$.
- The records kept in T_i's private undo log for $D^{invar}_{critical,i}$ are flushed to become the persistent $CriticalUndoLog$.
- Force data $\in D^{invar}_{critical,i}$ to non-volatile RAM.
- T_i releases all the locks it holds.

- Add T_i to the commit list.

Transaction T_i actually commits when its id is added to the commit list. After this has occurred, the system executes the following post-commit processing steps:

Post-commit Processing:

- Remove T_i from the list of active transactions.
- Discard persistent $CriticalUndoLog$.

5.2.1 Discussion

Strict two-phase locking guarantees that the commit order is the same as the serialization order. Although we write out log records in commit order, the order of transactions in the critical redo logs may not be the same as their serialization order. This is because we place a transaction's log records on a persistent log according to its LVI. However, because we assume the LVIs associated with any two values of a data created by two committed transactions are consistent with the transaction commit order, we can still ensure that each restored value is the value created by the last committed transaction. This can be done by recovering log records in the LVI order. Hence, our logging schemes will preserve execution correctness at recovery time. An important property of our logging schemes described above is that the overheads for our commit processing are bounded and reduced due to the following factors: (1) transactions pre-declare their data needs and (2) non-volatile RAM is used as the stable storage for critical data.

5.3 Failure Recovery

The recovery algorithm is executed on restart after a system crash, before the start of transaction processing. An important property of our recovery algorithm is that after a system crash, only a single pass over each log is needed. Besides, the recovery time for critical data is bounded.

5.3.1 Recovering critical variant data

For critical variant data, our recovery algorithm tries to recover temporally valid data from the persistent redo logs. Temporally invalid data will be refreshed by appropriate transactions and/or sensor readings. In the following, we describe an iterative checking algorithm used to determine which critical redo logs contain temporally valid data. Initially, we assume all critical redo logs contain temporally valid data. We then calculate the total time needed to recover all these logs, denoted as total_recovery_time. If the sum of current_time and total_recovery_time is smaller than $ELVI_1$ (i.e., earliest "last valid time instant" for $CriticalRedoLog_1$), then all critical redo logs contain temporally valid data and can thus be used for recovery. However, if the sum of current_time and total_recovery_time is grater than $ELVI_j$, but smaller than $ELVI_{j+1}$ for some $j > 1$, then this means that $CriticalRedoLog_1$, $CriticalRedoLog_2$, ..., $CriticalRedoLog_j$ contains temporally invalid data. Hence, these data must not be restored, but be refreshed. We then calculate the total time needed to refresh data on $CriticalRedoLog_1$, $CriticalRedoLog_2$, ..., $CriticalRedoLog_j$ and to restore data on $CriticalRedoLog_{j+1}$, $CriticalRedoLog_{j+2}$, ... We again check which logs will be invalid in this case and adjust the calculation of total_recovery_time if necessary. The pseudo-code for the algorithm is shown in Figure 2.

continue = TRUE;

/* $resB$ and $resE$ delimit the range of redo logs to be restored; initially we assume all logs can be restored */

$resB$ = 1; $resE$ = Num_CriticalRedoLogs;

/* $refB$ and $refE$ delimit the range of redo logs to be refreshed

$refB$ = 1; $refE$ = 0;

while (continue) {

restore_time = $\sum_{j=resB}^{resE} restime\_CriticalRedoLog_j$;

refresh_time = $\sum_{j=refB}^{refE} reftime\_CriticalRedoLog_j$;

total_recovery_time = restore_time + refresh_time;

current_time = wall_clock_time;

/* checking_time is the time needed to perform the following conditional statement and the while statement to exit the loop */

if (current_time + checking_time + total_recovery_time $\leq ELVI_{resB}$)

continue = FALSE;

else if ($ELVI_j$ < total_recovery_time < $ELVI_{j+1}$)

and (j < Num_CriticalRedoLogs) {

$refE = j$;

$resB = j + 1$;

if ($resB$ > Num_CriticalRedoLogs)

continue = FALSE;

}

}

Figure 2: Iterative algorithm for identifying logged data that need to be restored or refreshed

One can see that when the loop shown in Figure 2 terminates, the recovery procedure knows that $CriticalRedoLog_i$, $resB \leq i \leq resE$ will contain temporally valid data and can thus be restored. On the other hand, $CriticalRedoLog_j$, $refB \leq j \leq refE$ will contain invalid data. The recovery procedure can execute appropriate transactions and/or sensor readings to refresh those data.

In order to determine what data should be restored or refreshed, we associate with each $CriticalRedoLog_j$ the following two variables: $restime\_CriticalRedoLog_j$ and $reftime\_CriticalRedoLog_j$. $restime\_CriticalRedoLog_j$ denotes the total time needed to restore all data logged on $CriticalRedoLog_j$. $reftime\_CriticalRedoLog_j$ denotes the total time needed to refresh all data logged on $CriticalRedoLog_j$. When a transaction T_i' private redo log is about to be flushed to disk, we first calculate total times needed to restore and refresh data stored on T_i's private redo log. These times are then accumulated into $restime\_CriticalRedoLog_j$ and $reftime\_CriticalRedoLog_j$, respectively, assuming that T_i's private redo log is determined to be appended to $CriticalRedoLog_j$.

5.3.2 Recovering critical invariant data and non-critical data

Recall that critical invariant data updated by a transaction is forced to non-volatile RAM when the transaction commits. When a crash occurs, there can be at most one undo log for such data in non-volatile RAM. At recovery time, we first retrieve critical data from non-volatile RAM. We then perform a backward scan of this undo log and restore data stored on the log to its before image. After the system finishes recovering critical data, the system can open up to the environment again and start to process new critical transactions. Recovery actions for non-critical data can be done in the background. Non-critical data updated by a transaction is forced to a disk at transaction commit time. At recovery time, such data can be recovered by fetching the data stored in the undo log for non-critical data.

5.3.3 Predictable recovery time for critical data

Critical variant data. The recovery time for critical variant data depends on two factors: the maximum number of log records that may be processed for recovery purpose and the time needed to execute the iterative checking algorithm given in Figure 2. Apropos the number of log records that may be processed, it is not hard to see that this number can not be greater than the number of log records that contain temporally valid data just before the iterative checking algorithm begins. Note that the number of log records that contain temporally valid data just before the iterative checking algorithm begins depends on the durations of the temporal validity intervals that are associated with data in the database, which in turn depends on "last valid time instant" assigned to data updated by each transaction. We let TVI_i be the duration of the temporal validity interval associated with data that may be updated by transaction T_i. $S^{fvi}_{critical}$ denotes the set of critical transactions that may update critical variant data. Let P_i be the execution frequency of T_i. NLR_i denotes the number of log records that can be created by each instance of T_i. Hence, for each $T_i \in S^{fvi}_{critical}$, at most $\lceil TVI_i/P_i \rceil \times NLR_i$ log records will contain valid data that are created by T_i. The following lemma can be obtained.

Lemma 1 *At recovery time, there can be at most* $\sum_{T_i \in S^{fvi}_{critical}} \lceil TVI_i/P_i \rceil \times NLR_i$ *log records that contain valid data in the redo logs.*

Note that the longest time needed to execute the iterative checking algorithm occurs when the loop executes Num_CriticalRedoLogs times, i.e., the number of redo logs to be refreshed increases from zero to Num_CriticalRedoLogs. Without loss of generality, we assume $DLVI_i = C$ for all $i \geq 1$. In other words, the validity time interval for data stored in each redo log is identical. The following lemma can be obtained.

Table 2: System Parameters and Default Settings

Parameter	Setting	Meaning
VariantDBSize	250	Number of pages for critical variant data
InvariantDBSize	250	Number of pages for critical invariant data
NonCriticalDBSize	500	Number of pages for non-critical data
MemBufferSize	500	Number of pages in memory buffer pool
DiskIOTime	8 ms	Time to perform a disk access
NVRAMIOTime	0.167 ms	Time to perform a NVRAM access

Lemma 2 *The maximum value for Num_CriticalRedoLogs is given by* $\lceil max_{T_i \in S^{fvi}_{critical}} TVI_i/C \rceil$.

Critical invariant data. Recovering critical invariant data involves backward processing the log records stored in the undo log. The maximum size of this undo log is determined by a critical transaction that may update largest number of critical data with infinite validity intervals. Because transactions pre-declare their data needs, we can bound the time for this recovery process.

6 Performance Evaluation

This section presents the experimental setup and the assumptions made in our experiments. Our simulation evaluates the behavior of our logging and recovery algorithms with and without system failures. In the first set of experiments, we study how logging overheads affect the system's run-time performance. The primary performance metric is Missed Deadline Percentage (MDP), i.e., the percentage of transactions that miss their deadlines. In our study, a transaction is aborted as soon as its deadline expires. This corresponds to a firm real-time transaction. In our experiments, transactions are assigned priorities based on the earliest deadline first (EDF) policy. The conflict resolution for concurrency control is *priority abort*, where the conflicting transaction with lower priority waits or gets aborted depending on whether it is the requester or holder of locks, respectively. In the second experiment, we introduce system failures and measure the effect of such failures on recovery time and percentage of valid data that is restored from the logs.

6.1 Parameters of Simulation Model

We model the database itself as a collection of data pages in memory. The database buffer pool is modeled as a set of pages each of which can contain a single data item. The database consists of three classes of data items: critical variant data, critical invariant data, and non-critical data. System settings are controlled by the parameters listed in Table 2.

Transaction characteristics are controlled by the parameters listed in Table 3. In Table 3, CTv stands for the class of critical transactions that update variant data. CTiv stands for the class of critical transactions that update invariant data. NCT stands for the class of non-critical transactions. $U(i, j)$ denotes a uniformly distributed random variable in the range $[i, j]$.

For each approach tested, 20 simulation runs with different random number seeds are conducted and performance statistics are collected and averaged over the 20 runs. It is these averages and 90% confidence intervals

Table 3: Transaction Parameters and Default Settings

Parameter	Setting		Meaning
	CTv	CTiv & NCT	
CompFactor	4 ms	4 ms	Computation time per data accessed
ProbUpdate	0.4	0.4	Probability that a data access is an update
Period	U(400,500) ms		Period for CTv
ArrivalRate		14 trans./sec.	Mean arrival rate for CTiv and NCT
Percentage	50%	20% for CTiv 30% for NCT	Percentage for different classes of transactions
Slack		U(6.0,8.0)	Slack for CTiv and NCT
Length	U(10,15)	U(10,15)	Number of data accessed by a transaction

which are plotted in the graphs. Each run in our simulation continues until 700 transactions are executed. With this number of transactions executed, performance results were observed to stabilize. The three different classes of transactions coexist in each simulation run. We use three parameters, *PercentCTv*, *PercentCTiv*, and *PercentNCT* to control transaction mixes. Transactions in CTiv and NCT enter the system with exponentially distributed interarrival times. On the other hand, whenever CTv transactions are to be generated, a period is first chosen uniformly from a specified range with lower and upper bounds. We then generate CTv transactions with arrival times, $0, p, 2p, \ldots$, where p is the chosen period. In order to run CTv transactions concurrently with CTiv and NCT transactions, we set the arrival times of CTv transactions to be upper bounded by the maximum of arrival times of all CTiv and NCT transactions.

The computation requirement for a transaction T is estimated as $C(T) = Length' \times CompFactor$, where $Length'$ is the actual number of data accessed by T. In other words, we assume for each data accessed by a transaction, a fixed amount of computation is needed by the transaction. The deadline of any transaction in CTv is equal to the end of the transaction's period. The deadline of any other transaction T is set using the following formula:

$$d(T) = a(T) + (1 + Slack) * C(T),$$

where $a(T)$ is the arrival time of T and *Slack* is a uniformly distributed random variable within a specified range.

For the logging and recovery algorithm described in Section 5, the run-time I/O overhead for a transaction can involve three components: the first is the time needed to read data from stable storage into memory; the second is the time needed to write log records; and the third is the time needed to force updated data to stable storage before a transaction commits. We assume that logging can be done with one I/O access. Also, we assume all critical data is memory-resident and the memory buffer is used for the I/O of non-critical data. Hence, the amount of I/O overhead for a critical transaction that update variant data is simply *NVRAMIOTime*, i.e., the time needed to flush log records. On the other hand, the amount of I/O overhead for a critical transaction that update invariant data is $NVRAMIOTime \times (1 + ProbUpdate \times Length')$. This overhead includes writing of log records and forcing of updated invariant data. Finally, the expected amount of I/O overhead for a non-critical transaction involves all three components for a transaction explained above: (1) time to read data from disk into memory: $DiskIOTime \times (1 - ProbUpdate) \times Length' \times (1 - \frac{MemBufferSize}{NonCriticalDBSize})^5$; (2) time to write log records: $DiskIOTime$; (3) time to force updated non-critical data to disk: $DiskIOTime \times ProbUpdate \times Length'$.

6.2 Logging Overheads

Logging activities incur extra run-time overheads while adding recovery capability. In Section 5.1.4, we have described how to bound these overheads for our approach. Now we study how logging overheads affect major run-time performance, i.e., the percentage of transactions that miss their deadlines. We examine this issue by varying the following parameters: (1) transaction arrival rate, and (2) I/O overheads. For each experiment we perform, we show separate results for each of the three different classes of transactions. In the graphs, the suffix "no" indicates no logging is used. Hence, it corresponds to the performance of the baseline policy of EDF. One can expect that this baseline performs better than our algorithm in MDP, but of course no logging value can be obtained with this baseline. The suffix "cn" is used to denote our logging algorithm, with emphasis on critical data placed in NVRAM. As we explained in Section 5, using NVRAM instead of disk to store critical data can reduce transaction commit time and contribute to predictable transaction execution times. We also study how much advantage we gain in terms of reducing MDP by placing critical data and its logs in NVRAM instead of disk. We use the suffix "cd" in the graphs to denote that critical data and its logs, together with non-critical data, are placed in disk.

6.2.1 Varying Transaction Arrival Rate

In this experiment we vary the transaction arrival rate from 10 trans/sec to 18 trans/sec in increments of 1. The other parameters have the base values given in Tables 2 and 3. Figure 3 shows the performance for the class of CTv transactions. Observe that for an arrival rate of 14 trans/sec the CTv-no (baseline of EDF) misses about 4% of the deadlines and the CTv-cn (our approach) misses about 6.5% of the deadlines. For an arrival rate of 18 trans/sec the CTv-no misses about 26.8% of the deadlines and the CTv-cn misses about 30.7% of the deadlines. These results indicate that logging in our approach has a small effect on missing the CTv deadlines while adding recovery capability. Also, observe that for an arrival rate of 18 trans/sec the CTv-cd (critical data placed on disk) misses about 35% of the deadlines. This means that the use of NVRAM for CTv transactions gives us a relatively small benefit over the use of disk. The reason why the difference here is not bigger is simply because there is not much logging activity incurred for each committing CTv in our algorithm, i.e., just one I/O access to flush log records as we assumed in Section 6.1. However, the difference will be much more significant for the CTiv as we discuss below.

Compared to the baseline of EDF, logging in our approach also has a small effect on missing the CTiv deadlines while adding recovery capability. This is illustrated in Figure 4. For example, for an arrival rate of 14 trans/sec the CTiv-no misses 6% of the deadlines and the CTiv-cn misses about 10% of the deadlines. Observe that as the load increases, CTiv-cn performs significantly better than CTiv-cd. For an arrival rate of 18 trans/sec the CTiv-cn misses about 43.7% of the deadlines, but the CTiv-cd misses about 62% of the deadlines. This is certainly due to our design that committing a CTiv transaction requires not only flushing log records but also flushing updated invariant data itself as well.

Figure 5 shows the performance for the class of NCT transactions. Logging overheads for our approach become significant in this case. Observe that the performance of the NCT-cn is close to that of the NCT-cd.

[5] $\frac{MemBufferSize}{NonCriticalDBSize}$ denotes the probability that a needed non-critical data is in the memory buffer.

This is because non-critical data is stored in disk in our algorithm (see Table 1). Note that the performance of the NCT-cn and NCT-cd is close, but not identical. This is because there are two other types of transactions concurrently executing in the system. In other words, this is because the CTv-cn and CTiv-cn miss smaller numbers of deadlines than the CTv-cd and CTiv-cd as shown in Figures 3 and 4.

We have also measured the overall MDP performance when all the transactions are put together on one graph, regardless of transaction and data types. The results show similar trends (not shown here due to space limit). Based on these simulation results, we conclude that our approach incurs small logging overhead which has small effect on missing transaction deadlines. This overhead paid off at the system recovery time to be discussed in Section 6.3.

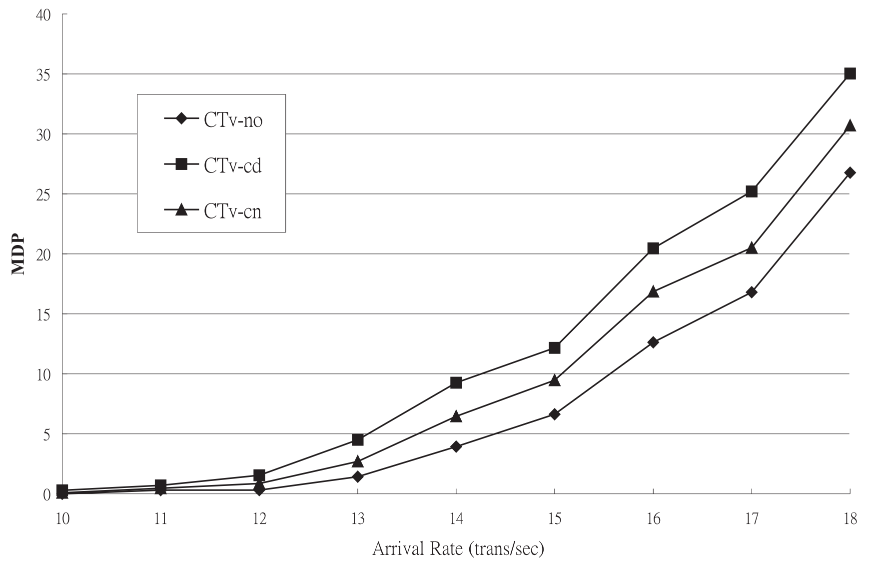

Figure 3: MDP vs Arrival Rate for critical transactions that update variant data

6.2.2 Varying I/O Overheads

Without question I/O overheads dominate logging activity. Based on the analysis given in [Jor], we have assumed one disk access requires 8 ms and one NVRAM access needs 0.167 ms as given in Table 2. In this experiment, we use these figures as base values and scale them up with a scale factor ranging from 1 to 5 in steps of 1. In effect, each particular scale factor amounts to a data page whose size is the product of the scale factor and the base page size. As Gray and Shenoy pointed out in [GS00], disk page sizes have grown from 2KB to 8KB over the last decade and are expected to increase 5 fold per decade. This experiment thus tests the adaptability of our approach to future storage device trends.

In this experiment, we fix ArrivalRate to be 14 trans/sec, i.e., the mean arrival rate for the class of CTiv and NCT transactions. Figures 6, 7, and 8 illustrate the results for the CTv, CTiv, and NCT, respectively. In each case, the MDP remains constant for the baseline of EDF, irrespective of what the page size is. This is certainly because the baseline does not do logging. Hence, no I/O overhead is incurred, but post-crash recovery is impossible. In Figures 6 and 7, as the page size increases, the MDP gradually increases for both the CTv-cn and CTiv-cn. For example, for an I/O Scale Factor of 3, the CTv-cn misses about 9% of the deadlines and the

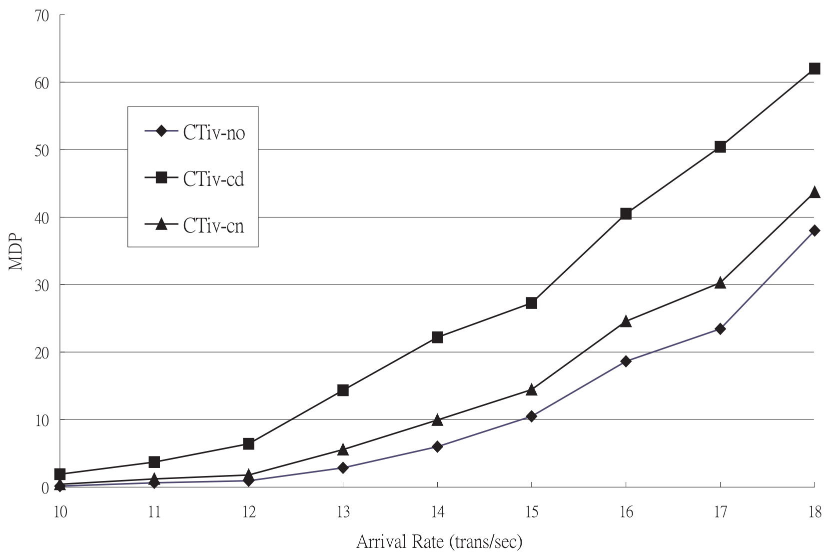

Figure 4: MDP vs Arrival Rate for critical transactions that update invariant data

CTiv-cn misses about 16% of the deadlines. For an I/O Scale Factor of 5, the CTv-cn misses about 10% of the deadlines and the CTiv-cn misses about 19% of the deadlines. Based on these results, we can infer that our approach can be adapted to future generations of storage devices without significant problems.

Observe also from Figure 6, the CTv-cd misses about 21% of the deadlines for an I/O Scale Factor of 5, 11% higher than that for the CTv-cn. The situation is even worse for the CTiv-cd; Figure 7 shows that the CTiv-cd misses about 94.5% of the deadlines for an I/O Scale Factor of 5, 75% higher than that for the CTiv-cn. These results tell us that in future generations of real-time databases, the use of NVRAM for logging critical transactions and data is necessary not only for the purpose of predictable transaction execution times, but also for the performance consideration of small MDP.

6.3 Recovery-Time Performance

In this section, we evaluate after-crash performance of our recovery algorithm by introducing a system failure during each simulation run. As before, all three classes of transactions co-exist in the system. During each simulation run, we inject a failure at a time point that is randomly selected from among the $2 \times TransNum$ event time points which represent the arrivals and commits of all generated transactions. In this experiment, we calculate the recovery time for our approach as a summation of the following terms: (1) the time needed to restore valid variant data calculated as $NVRAMIOTime \times Num\_Valid\_Variant\_Data$; (2) the time needed to refresh or recompute remaining variant data determined as $CompFactor \times Num\_Remaining\_Variant\_Data$[6]; (3) the time needed to restore invariant data calculated as $NVRAMIOTime \times MaxLength$, where $MaxLength$ is the maximum number of data accessed by a transaction. For comparison purpose, we also compute the recovery time for a traditional recovery technique. The traditional recovery technique we compare with does not distinguish critical data as either valid or not. Hence, all critical data will be refreshed or recomputed. Furthermore, all

[6] We assume for each data to be refreshed or recomputed, a fixed amount of computation is needed. The same assumption was made in Section 6.1 for estimating transaction computation requirements.

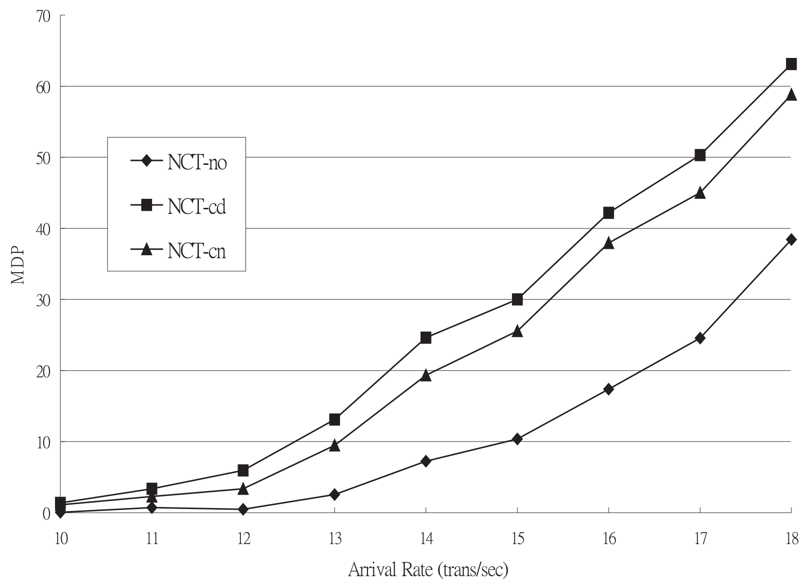

Figure 5: MDP vs Arrival Rate for non-critical transactions

data including both critical and non-critical data must be restored before the system can restart its operation.

Figure 9 illustrates the comparison results for $MaxPeriod = 500$ ms. Along the y-axis, 1p, 2p, 3p, and 4p denote the durations of temporal validity interval. TR stands for the traditional recovery technique we compare with. VVD denotes the recovery time for valid variant data. RVD denotes the recovery time for remaining variant data. IVD denotes the recovery time for invariant data. NCD denotes the recovery time for non-critical data. Because the traditional approach we compare with does not distinguish critical data as valid or not and hence refreshes all critical data, we show its recovery time for critical data as an IVD. Further, the traditional approach must restore non-critical data before the system resumes its major operation, its recovery time also includes a NCD. On the other hand, our approach includes a VVD, a RVD, and an IVD, but not NCD. Because IVD is much smaller than other recovery times (i.e., VVD and RVD) in our approach, the graph does not show IVD portions for our approach.

One can see that as the duration of temporal validity interval increases, total recovery time for our approach becomes shorter. For example, in Figure 9, for a duration of temporal validity interval equal to 1p, the recovery time is 860 ms. For a duration of temporal validity interval equal to 4p, the recovery time becomes 387 ms. Similar results have been observed for other max periods (not shown here due to space limit). With the parameters given in Tables 2 and 3, our simulation results show that restoring more variant data from the logs and refreshing remaining variant data takes less time. The simulation results also show that the traditional approach requires 2400 ms for complete recovery. Hence, recovery using our approach is 3 to 6 times faster than the traditional recovery technique.

7 Conclusion

This paper has proposed three principles for fault tolerance in real-time databases. The principles are distinct from those for conventional databases in that timing predictability and temporal validity are the primary design

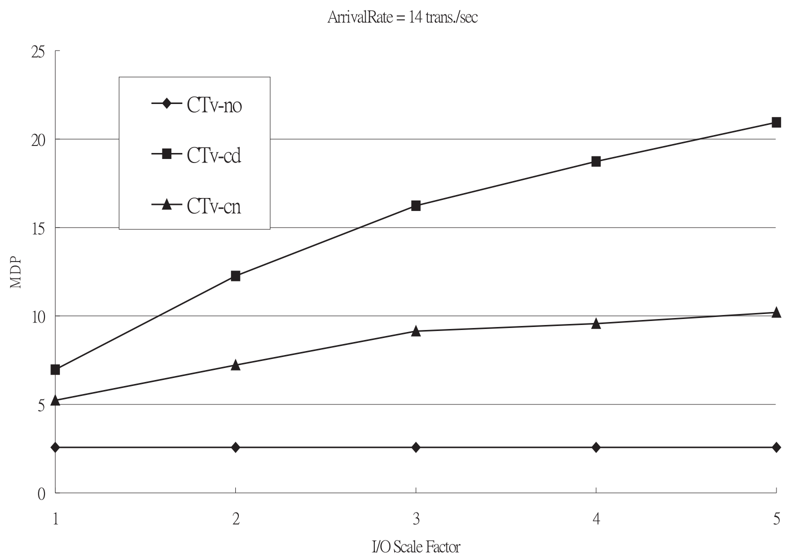

Figure 6: MDP vs I/O Scale Factor for CTv transactions

criteria. Based on these principles, we propose a real-time logging and failure recovery algorithm that supports these principles and is well suited to an important class of real-time database applications. The algorithm has some important properties, e.g., it minimizes normal runtime overhead caused by logging and has a predictable impact on transaction timing constraints. Upon a failure, the system can recover critical data to a consistent and temporally valid state within predictable time bounds. The system can then resume its major functioning, while non-critical data is being recovered in the background. As a result, the recovery time is bounded and shortened. Our performance evaluation via simulation shows that logging overhead has a small effect on missing transaction deadlines while adding recovery capability. Experiments also show that recovery using our approach is 3 to 6 times faster than traditional recovery.

References

[CKKS89] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *Proc. 15th International Conference on Very Large Data Bases*, 1989.

[GS00] Jim Gray and Prashant Shenoy. Rules of thumb in data engineering. In *Proc. IEEE International Conference on Data Engineering*, 2000.

[Jor] John Jory. The state of solid state disk. http://www.imperialtech.com/technology_whitepapers_statessd.htm.

[KH98] Vijay Kumar and Meichun Hsu, editors. *Recovery Mechanisms in Database Systems.* Prentice Hall PTR, 1998.

[KS98] Vijay Kumar and Sang H. Son. *Database Recovery.* Kluwer International, 1998.

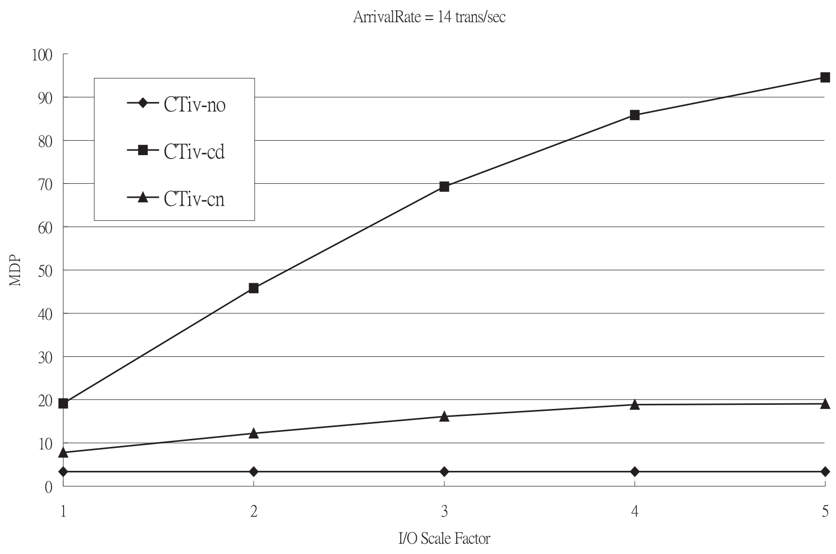

Figure 7: MDP vs I/O Scale Factor for CTiv transactions

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwartz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), March 1992.

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

[SRS97] R. M. Sivasankaran, K. Ramamritham, and J. A. Stankovic. Logging and Recovery Algorithm for Real-time Database. Technical report, Dept. of Computer Science, Univ. of Massachusetts, 1997.

[SRST95] R.M. Sivasankaran, K. Ramamritham, J. Stankovic, and D. Towsley. Data Placement, Logging and Recovery in Real-Time Active Databases. In *Proc. International workshop on active and real-time database systems*, June 1995.

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

[SSK99] LihChyun Shu, Huey-Min Sun, and Tei-Wei Kuo. Shadowing-based crash recovery schemes for real-time database systems. In *Proc. 11th Euromicro Conference on Real-Time Systems*, pages 260–267, 1999.

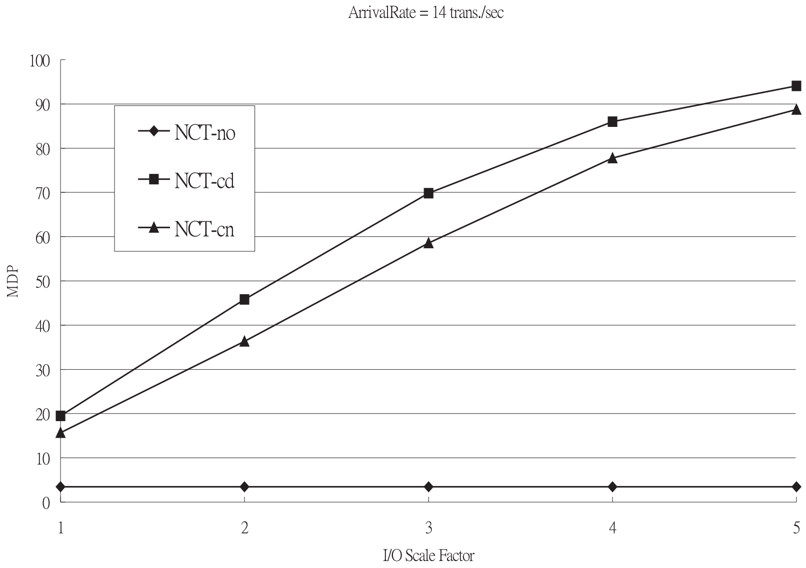

Figure 8: MDP vs I/O Scale Factor for NCT transactions

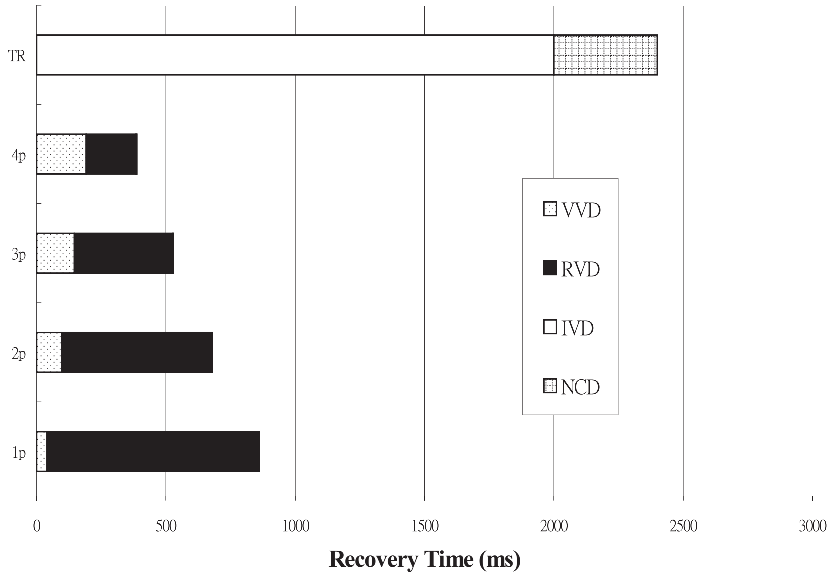

Figure 9: Recovery time for our approach and traditional approach with $Max\_Period = 500$ ms