Two-Version-Based Concurrency Control and Recovery in Real-Time Client/Server Databases[*][†]

Tei-Wei Kuo, Yuan-Ting Kao[†], and Chin-Fu Kuo
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106, ROC
[†]Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan 621, ROC

Abstract

While there has been a significant amount of research in real-time concurrency control, little work has been done in logging and recovery for real-time databases. This paper proposes a two-version approach which considers both real-time concurrency control and recovery. We propose a network-server-based architecture and algorithms which can not only reduce the blocking time of higher-priority transactions and improve the response time of client-side read-only transactions but also provide a diskless run-time logging mechanism and an efficient and predictable recovery procedure. The performance of the algorithms was verified by a series of simulation experiments by comparing the algorithms with the well-known Priority Ceiling Protocol (PCP), the Read/Write PCP, the New PCP, and the 2-version two-phase locking protocol, for which we have very encouraging results. The schedulability of higher-priority transactions and the response time of client-side read-only transactions were all greatly improved.

KEY WORDS AND PHRASES: Real-Time Database, Concurrency Control, Recovery, Read-Only Transactions, Client/Server Databases, Logging, Write Through Procedure.

[*]Supported in part by research grants from the National Science Council under Grants NSC87-2213-E-194-018 and NSC87-2213-E-309-002

[†]This paper is an extended version of a paper in the *IEEE Third International High Assurance Systems Engineering Symposium*. Yuan-Ting Kao graduated from the National Chung Cheng University under the thesis supervision of Tei-Wei Kuo.

1 Introduction

Real-time concurrency control has been an active research topic in the past decades. A number of researchers, [5, 6, 7, 14, 16, 21, 17, 22, 18, 26, 27, 25, 30, 32], have proposed various effective mechanisms in the concurrency control of real-time data access. In particular, semantics-based concurrency control, [6, 7, 16, 19, 26, 27, 25, 32], has been shown to improve the system schedulability significantly. Researchers also explored issues in processing read-only transactions, such as those [18, 19] based on the idea of dynamic adjustment of serializability order [22].

As issues in real-time concurrency control are better understood, the demand of system reliability is increasing. Although a lot of research, [8, 12, 11, 24], has been done in logging and recovery for traditional databases, little work explores logging and recovery for real-time databases [13, 31]. Rare research considers issues related to both real-time concurrency control and durability issues. In particular, Gupta, Haritsa, and Ramamritham [10] proposed a new real-time commit protocol which allows transactions to "optimistically" borrow uncommitted data in a controlled manner to minimize the number of deadline violations. Sivasankaran, Ramamritham, and Stankovic [31] proposed a partitioned logging and recovery algorithm for real-time disk-resident databases. The log is partitioned according to data classes, such as critical and temporal ones, to provide parallel logging and recovery. Non-volatile RAM-based devices are used to reduce the unpredictability of real-time databases. Huang and Gruenwald [13] proposed a checkpointing technique for real-time main memory databases. A database is partitioned according to data types (persistent type vs. temporal type) and update frequencies. The system checkpoints each partition independently based on its update frequency and its temporal valid interval.

While little work has been done in logging and recovery for real-time databases, the close relationship between real-time concurrency control and recovery (and logging) has been ignored in the past decade. Note that a schedule is recoverable only if no transaction τ commits before any transaction from which τ reads commits. A real-time concurrency control protocol should ensure that conflicting transactions commit in the order of their read-from relationship. This paper proposes an integrated mechanism for concurrency control and recovery in real-time databases. A two-version-based concurrency control protocol called *Two-Version Priority Ceiling Protocol* (2VPCP), is proposed to reduce the blocking time of higher-priority transactions based on the idea of dynamic serializability adjustment [17, 19, 22] without relying on local data updates for transactions [17, 19, 22]. The 2VPCP protocol is, then, extended to a distributed environment to process read-only transactions at client-side systems locally. The resulting system can not only significantly boost the response time of read-only transactions issued at client-side systems, but also virtually eliminate the interference of conflicting data accesses between client-side read-only transactions and server-side transactions. The extended 2VPCP protocol not only associates each client-side system with a consistent database image for local processing of read-only transactions, but also provides an efficient recovery mechanism. The performance of the algorithms was verified by a series of simulation experiments, for which we had very encouraging results. Comparisons of different recovery mechanisms are also presented to demonstrate the capability of the two-version approach.

There are two major contributions in this paper: (1) The effectiveness of the two-version approach is shown in reducing the blocking time of higher-priority transactions and in improving the response time of client-side read-only transactions. Note that the results of this paper are orthogonal to any previous research in processing read-only transactions, [18, 19], which consider weaker correctness criteria or access patterns of transactions. All transactions in our system are serializable. (2) A two-version (network-server-based) architecture is proposed to not only support a diskless run-time logging mechanism and an effective write-through procedure, but also provide an efficient and predictable recovery mechanism. The logging mechanism and an effective write-through procedure virtually have no impact on the executions of transactions in the system.

The rest of this paper is organized as follows: Section 2 extends the Read/Write Priority Ceiling Protocol (RWPCP) [30] into a two-version-based protocol called the Two-Version Priority Ceiling Protocol (2VPCP). The properties of the 2VPCP protocol are then proven. Section 3 further extends the 2VPCP protocol to a distributed environment to locally and efficiently process read-only transactions at client-side systems. The correctness of the extended protocol is proven. Section 4 proposes an efficient and predictable recovery mechanism based on the extended 2VPCP protocol. Section 5 provides experimental results which demonstrate the performance of the algorithms. Section 6 is the conclusion.

2 The 2VPCP Protocol

2.1 Overview

The Read/Write Priority Ceiling Protocol (RWPCP) [30] has shown the effectiveness of using read and write semantics in improving the performance of the Priority Ceiling Protocol (PCP) [29] in real-time concurrency control. While PCP only allows exclusive locks on data objects, RWPCP introduces a write priority ceiling WPL_i and an absolute priority ceiling APL_i for each data object O_i to emulate share and exclusive locks, respectively. The write priority ceiling WPL_i of data object O_i is equal to the highest priority of transactions which may write O_i. The absolute priority ceiling APL_i of data object O_i is equal to the highest priority of transactions which may read or write O_i. When data object O_i is read-locked, the read/write priority ceiling $RWPL_i$ of O_i is equal to WPL_i. When data object O_i is write-locked, the read/write priority ceiling $RWPL_i$ of O_i is equal to APL_i. A transaction instance may lock a data object if its priority is higher than the highest read/write priority ceiling $RWPL_i$ of the data objects locked by other transaction instances. When a data object O_i is write-locked, the setting of $RWPL_i$ prevents any other transaction instance from write-locking O_i because $RWPL_i$ is equal to APL_i. When a data object O_i is read-locked, the setting of $RWPL_i$ only allows a transaction instance with a sufficiently high priority to read-lock O_i in order to constrain the number of priority inversions for any transaction instance which may write-lock O_i because $RWPL_i$ is equal to WPL_i.

Lam and Hung [17] further sharpened the RWPCP by proposing the idea of dynamic adjustment of serializability order for hard real-time transactions, where Lin and Son [22] proposed the idea of dynamic adjustment of serializability order for optimistic real-time concurrency control. With a delayed write procedure, a higher-priority transaction instance may preempt a lower-priority transaction instance by using the Thomas Write rules when a write-write conflict exists, where a delayed write procedure requires every transaction instance to only update data objects in its local space and to delay the updating of the database until the commitment of the transaction instance. The read-write conflict between conflicting transaction instances is partially resolved by allowing a higher-priority transaction instance to read the database even though a lower-priority transaction instance has write-locked the data object. Note that the delayed write procedure requires every transaction instance to only update data objects in its local space, and the above preemption in read-write conflict lets the higher-priority transaction instance precede the lower-priority transaction instance in the serializability order.

Although the new protocol introduced by Lam and Hung [17] significantly reduces the blocking time of higher-priority transaction instances under RWPCP, every transaction instance may need extra space to keep its own local copy for each of its updated data objects because of the delayed write procedure. On the other hand, only the response time of higher-priority transactions is improved, and the executions of read-only transaction instances tend to be serialized. Little work, including [17, 30], has been done in considering recovery when concurrency control protocols are proposed.

req\locked	R	W	C
R	yes	yes	no
W	yes	no	no
C	no	no	no

Table 1: The compatibility matrix of locks.

This paper proposes a two-version approach which considers both real-time concurrency control and recovery. We propose to use the idea of two-version databases to replace a delayed write procedure to save the extra space needed by the procedure. The goal is to first propose a two-version variation of the RWPCP [30] to have the flexibility in the dynamic adjustment of transaction serializability order to favor higher-priority transactions and read-only transactions. We will then extend the protocol and the idea of two-version databases into distributed environments for efficient and local processing of read-only transactions and provide efficient and predictable failure recovery. Note that little work has been done for concurrency control in distributed real-time environments, e.g., [20, 15, 28]. Since there can be a large number of read-only transactions in many commercial database systems, how to improve the response time of read-only transactions is of paramount importance.

We assume that a transaction system consists of a fixed set of transactions. (This condition will be relaxed when local processing of read-only transactions is considered in Section 3.) Each data object has two versions: a consistent version and a working version, where the consistent version contains a data value updated by a committed transaction instance, and the working version contains a data value updated by an uncommitted transaction instance. There are three kinds of locks in the system: read, write, and certify. Before a transaction reads (or writes) a data object, it must first read-lock (or write-lock) the data object. A read operation on a data object always reads from the consistent version of the data object. A write operation on a data object always updates the working version of the data object. It is required that, before a transaction commits, the transactions must transform each of its write locks into a certify lock on the same data object. As soon as a transaction obtains a certify lock on a data object, it can copy its updated working version of the data object to the consistent version. There is no requirement on the order or timing of lock transformations. The transformation of a write-lock into a certify-lock is considered as requesting a new certify lock. If the request of a certify-lock by a transaction instance is not granted, the transaction is blocked by the system until the request is granted. When a transaction terminates, it must release all of its locks. The compatibility matrix of locks is shown in Table 1. (The well-known Two-Version Two-Phase Locking scheme has the same compatibility matrix [4, 9]. Note that the Two-Version Two-Phase Locking scheme could not guarantee one priority inversion for real-time transactions and may suffer from the deadlock problem.) A certify lock is stronger than a write lock, and a write lock is stronger than a read lock. All transactions follow the two-phase locking (2PL) scheme. The details will be shown in later sections. Compared to the Read/Write Priority Ceiling Protocol (RWPCP) [30], the two-version locking mechanism could provide higher-priority (and read-only) transactions better opportunities to preempt lower-priority transactions. However, it would be at the cost of extra certify locks and the copying of the updated working version of data objects to the consistent version. The number of certify locks which must be obtained by a committing transaction is the same as the number of write locks already obtained by the committing transaction. The cost in copying the updated working version of data objects to the consistent version is also proportional to the number of write locks already obtained by the committing transaction.

Now, we will state our notation.

Notation:

- $\tau_{i,j}$ denotes the j_{th} instance of transaction τ_i. p_i and c_i are the period and worst-case computation time of transaction τ_i, respectively. If transaction τ_i is aperiodic, p_i is the minimal separation time between its consecutive requests. When there is no ambiguity, we use the terms "transaction" and "transaction instance" interchangeably.

- $R_{i,j}$ denotes the j_{th} request of transaction τ_i. A transaction instance $\tau_{i,j}$ is initiated for each request of transaction τ_i. Once transaction instance $\tau_{i,j}$ is aborted, $\tau_{i,j}$ may be restarted or terminated, as required by the selected scheduling algorithm.

- The k_{th} critical section of a transaction instance $\tau_{i,j}$ is denoted as $z_{i,j,k}$ and corresponds to the code segment between the k_{th} locking operation and its corresponding unlocking operation. We assume in this paper that critical sections are properly nested. In other words, if the locking operation of a semaphore is no later than the locking operation of another semaphore within a transaction instance, the corresponding unlocking operation of the former semaphore is no earlier than the corresponding unlocking operation of the later semaphore. Note that it is one of the assumptions of PCP in handling the priority inversion problem.

- $W(O_i)$ and $C(O_i)$ denote the working version and consistent version of data object O_i, respectively.

2.2 The Basic 2VPCP Protocol

The *Two-Version Priority Ceiling Protocol* (2VPCP) is a two-version variation of the Read/Write Priority Ceiling Protocol [30]. The rationale behind the design of the 2VPCP protocol is to have flexibility in the adjustment of transaction serializability order to favor higher-priority transactions and read-only transactions. In later sections, we shall then extend the 2VPCP protocol into distributed environments for local processing of read-only transactions and efficient failure recovery.

In this section, we are interested in the context of uniprocessor priority-driven preemptive scheduling, and every transaction has a fixed priority. (This condition will be relaxed when local processing of read-only transactions is considered in Section 3.) The real-time database can be either memory-resident or disk-resident. As defined in [30], the write priority ceiling WPL_i of data object O_i is equal to the highest priority of transactions which may write O_i. The absolute priority ceiling APL_i of data object O_i is equal to the highest priority of transactions which may read or write O_i. Since 2VPCP adopts a two-data-version approach and introduces a new lock called *certify lock*, the setting of the read/write priority ceiling $RWPL_i$ of each data object O_i is modified as follows: The read/write priority ceiling $RWPL_i$ of each data object O_i is set dynamically. When a transaction read-locks or write-locks O_i, $RWPL_i$ is equal to WPL_i. When a transaction certify-locks O_i, $RWPL_i$ is equal to APL_i. Note that any read operation on a data object always reads from the consistent version of the data object, and any write operation on a data object always writes into the working version of the data object. It is required that, before a transaction commits, the transactions must transform each of its write locks into a certify lock on the same data object. A certify lock on a data object secures the copying of the data value from its working version into the consistent version. No lock transformation is required for a read lock.

The rationale behind the setting of priority ceilings is as follows: When data object O_i is write-locked, $RWPL_i$ is set as WPL_i to prevent any subsequent transaction from write-locking O_i because WPL_i is equal to the highest priority of transactions which may write O_i. Note that there is only one working version for each data object. When data object O_i is certify-locked, $RWPL_i$ is set as APL_i so that no other subsequent transactions can lock O_i in any mode. This is to secure the copying of the data value from the working version of O_i into its consistent version. When data object O_i is read-locked, $RWPL_i$ is set as WPL_i so that only transactions which have a priority higher than WPL_i can read-lock O_i afterward. This constraint is to prevent any transaction which might later write-lock O_i from being blocked by more than one lower-priority transaction which read-locks O_i.

We now present the definition of 2VPCP:

1. A transaction instance, which has the highest priority among all ready transaction instances, is assigned the processor. If a transaction instance does not attempt to lock any data object, the transaction instance can preempt the execution of any transaction instance with a lower priority, whether or not the priorities are assigned or inherited. (Priority inheritance will be defined later.)

2. When a transaction instance $\tau_{i,j}$ attempts to read-lock, write-lock, or certify-lock a data object O_k, the priority of $\tau_{i,j}$ must be higher than the read/write priority ceilings of all data objects currently locked by transaction instances other than $\tau_{i,j}$; otherwise, the lock request is blocked. If the priority of $\tau_{i,j}$ is higher than the read/write priority ceilings of all data objects currently locked by transaction instances other than $\tau_{i,j}$, there are three cases to consider:

   (a) If $\tau_{i,j}$ requests a read lock on O_k, then $\tau_{i,j}$ read-locks O_k, and the read/write priority ceiling $RWPL_k$ of data object O_k is set as WPL_k.

   (b) If $\tau_{i,j}$ requests a write lock on O_k, then $\tau_{i,j}$ write-locks O_k, and the read/write priority ceiling $RWPL_k$ of data object O_k is set as WPL_k.

   (c) If $\tau_{i,j}$ requests a certify lock on O_k, then $\tau_{i,j}$ certify-locks O_k, and the read/write priority ceiling $RWPL_k$ of data object O_k is set as APL_k. Note that $\tau_{i,j}$ must have write-locked O_k before it requests a certify lock on O_k, and both APL_k and WPL_k are no less than the priority of $\tau_{i,j}$.

   If the priority of $\tau_{i,j}$ is no higher than the read/write priority ceilings of all data objects currently locked by transaction instances other than $\tau_{i,j}$, then the lock request is blocked. Let O^* be the data object with the highest read/write priority ceiling of all data objects currently locked by transaction instances other than $\tau_{i,j}$. If $\tau_{i,j}$ is blocked because of O^*, $\tau_{i,j}$ is said to be blocked by the transaction instance that locked O^*.

3. A transaction instance $\tau_{i,j}$ uses its assigned priority, unless it locks some data objects and blocks higher priority transaction instances. If a transaction instance blocks a higher priority transaction instance, it inherits the highest priority of the transaction instances blocked by $\tau_{i,j}$. When a transaction instance unlocks a data object, it resumes the priority it had at the point of obtaining the lock on the data object. When a transaction instance is aborted, all transaction instances which inherit its priority must reset their priorities according to the definition of priority inheritance. The priority inheritance is transitive. Note that the resetting of priority inheritance

can be efficiently implemented by using a stack data structure. We refer interested readers to [29] for details. This is because there is no transitive blocking in transaction executions (Please see Lemma 2).

4. All transaction instances follow a 2PL scheme. That is, no transaction instance is allowed to obtain any new lock after it releases any locks.

The lock compatibility matrix, as shown in Table 1, is implicitly verified through priority ceilings (Please see Lemmas 3, 4, and 5). A transaction instance is before another transaction instance in the serializability order if any of the following conditions is satisfied: (a) the latter reads from the consistent version of any data object updated by the former. (b) the former and the latter update the same data object, and the former write-locks the data object first. Theorem 4 shows that all 2VPCP schedules are serializable. Theorem 5 also shows that the serializability order of transaction instances is the same as their *begin_unlock* message order. The priority ceilings are used to control priority inversion in the system. A similar approach can be found in [30]. The aborting of a transaction may happen because of its deadline violation. However, transaction aborting incurs low overheads because executing transactions are updating only the working version of any data object. When a transaction commits, the transaction transforms each of its write locks into a certify lock on the same data object and copies its updated working version of the data object to the consistent version. The aborting of a transaction simply discards the working version, and any subsequent transaction can simply overwrite the working version. Since all transactions read from the consistent version of data objects, no cascading aborting is possible. Furthermore, it is not possible for the cascaded resetting of any inherited priority for any transaction due to the occurrences of transaction abortings because there is no transitive blocking, as shown in Lemma 2 in the next section.

Example 1 *A 2VPCP Schedule:*

We illustrate the 2VPCP protocol by an example. Suppose that there are three transactions τ_1, τ_2, and τ_3 in a uniprocessor environment. Let the priorities of τ_1, τ_2, and τ_3 be 1, 2, and 3, respectively, where 1 is the highest, and 3 is the lowest. Suppose that τ_1 and τ_2 may read and write data object S_1, respectively, and τ_2 and τ_3 may read and write data object S_2, respectively. According to the definitions of ceilings, the write priority ceiling WPL_1 and the absolute priority ceiling APL_1 of S_1 are 2 and 1, respectively. The write priority ceiling WPL_2 and the absolute priority ceiling APL_2 of S_2 are 3 and 2, respectively.

At time 0, τ_3 starts execution. At time 2, τ_3 write-locks S_2 successfully, and $RWPL_2 = WPL_2 = 3$. At time 4, τ_2 arrives and preempts τ_3. At time 6, τ_2 write-locks S_1 success-

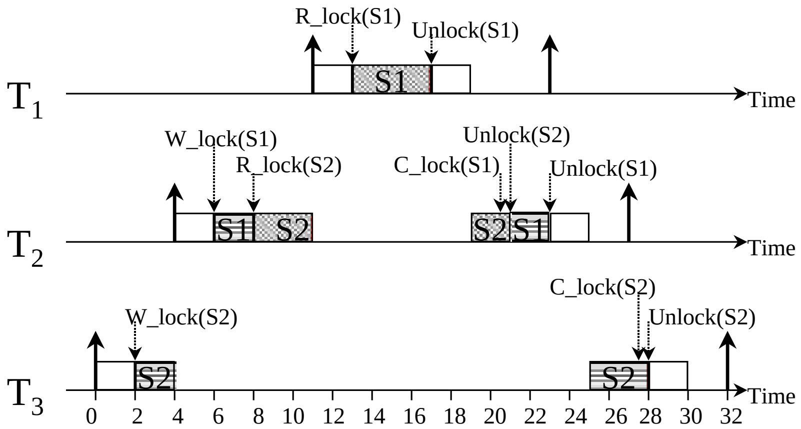


Figure 1: A 2VPCP schedule

fully because the priority of τ_2 is higher than $RWPL_2$ ($RWPL_1 = WPL_1 = 2$). At time 8, τ_2 read-locks S_2 successfully because the priority of τ_2 is higher than $RWPL_2$ ($RWPL_2 = WPL_2 = 3$). Note that τ_3 is behind τ_2 in the serializability order although τ_3 write-locks S_2 before τ_2 read-locks S_2. At time 11, τ_1 arrives and preempts τ_2. At time 13, τ_1 read-locks S_1 successfully because the priority of τ_2 is higher than $RWPL_1$ and $RWPL_2$. $RWPL_1$ is equal to $WPL_1 = 2$. Note that τ_2 is behind τ_1 in the serializability order although τ_2 write-locks S_1 before τ_1 read-locks S_1. τ_1 then unlocks $S1$ and commits at time 17 and 19, respectively. Right before time 21, τ_2 certify-locks S_1 successfully and copies the working version of $S1$ into the consistent version because the priority of τ_2 is higher than $RWPL_2$ ($RWPL_2 = WPL_2 = 3$). At time 21, τ_2 unlocks S_2. At time 23, τ_2 unlocks S_1. At time 25, τ_2 commits, and τ_3 resumes its execution. Right before time 28, τ_3 certify-locks S_2 successfully and copies the working version of $S2$ into the consistent version. At time 30, τ_3 commits.

For comparison, let us schedule these transactions according to the Read/Write Priority Ceiling Protocol (RWPCP), where a single version per data object is considered. As shown in Figure 2, the write-lock request of τ_2 on S_1 is rejected at time 6 because the priority of τ_2 is no higher than $RWPL_2 = APL_2 = 2$. The reason for the rejection under the RWPCP protocol is because τ_2 may later read S_2, and the read will leave τ_2 behind τ_3 in the serializability order. As a result, τ_2 is blocked by τ_3. Note that the 2VPCP protocol lets τ_2 preempt τ_3, and τ_2 reads from the consistent version of S_2. As a result, τ_2 is not blocked by τ_3 under the 2VPCP protocol. Figure 2 also shows the blocking of τ_1 at time 13 under the RWPCP protocol.

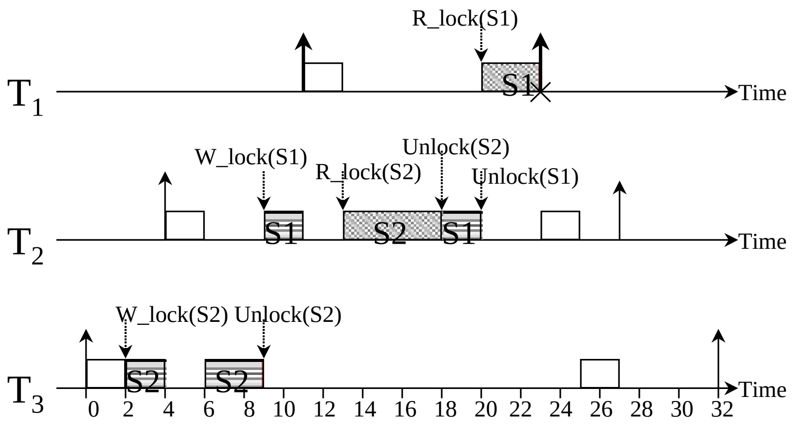


Figure 2: A RWPCP schedule

This example demonstrates that one of the goals in designing the 2VPCP protocol is that a higher-priority transaction instance can utilize the consistent version of a data object without being blocked by a lower-priority transaction instance, due to read/write conflicts. The serializability order of transaction instances is no longer determined by the order of the conflicting lock requests and can be adjusted according to the priorities of the transaction instances. □

2.3 Properties

Lemma 1 *A (higher-priority) transaction instance τ_H can be blocked by another (lower-priority) transaction instance τ_L only if τ_L is executing in a critical section which later blocks τ_H (when τ_H is initiated).*

Proof According to the definitions of the 2VPCP protocol, τ_L can block τ_H only if τ_L directly blocks τ_H because of a lock request, or τ_L inherits a priority higher than the priority of τ_H. In either case, τ_L must be in a critical section to block τ_H. Furthermore, if τ_L is not executing in a critical section when τ_H is initiated, then τ_H can preempt τ_L because its priority must be no more than the priority of τ_H. □

Definition 1 *[29] Transitive blocking is said to occur if a (higher priority) transaction instance is directly blocked by another (lower priority) transaction instance which, in turn, is directly blocked by the other (further lower priority) transaction instance.*

Lemma 2 *No transitive blocking is possible.*

Proof. This lemma can be proven by contradiction. Suppose that a transitive blocking happens among three distinct transaction instances τ_1, τ_2, and τ_3. Let τ_1 directly block τ_2, and τ_2 directly block τ_3. According to Lemma 1, τ_1 and τ_2 must be executing in critical sections to block τ_2 and τ_3, respectively. Let τ_1 and τ_2 be executing in critical sections $z_{1,i}$ and $z_{2,j}$, respectively, when the transitive blocking occurs. Since τ_1 blocks τ_2, τ_1 must enter critical section $z_{1,i}$ before τ_2 is initiated; otherwise, τ_2 will not be directly blocked by τ_1. By the definitions of the 2VPCP protocol, the read/write priority ceiling $RWPL_k$ of the data object O_k locked by τ_1 when τ_1 enters $z_{1,i}$ should be no lower than the priority of τ_2. However, when τ_2 requests a lock to enter critical section $z_{2,j}$ (which blocks τ_3 later), its priority should still be no larger than the read/write priority ceiling $RWPL_k$ of the data object O_k. In other words, τ_2 will not be allowed to enter critical section $z_{2,j}$ until τ_1 leaves $z_{1,i}$, and the transitive blocking should not occur. $\Box$

Theorem 1 *2VPCP is deadlock-free.*

Proof. Since there is no transitive blocking (please see Lemma 2), a deadlock can only happen between two transaction instances. Let two distinct transaction instances τ_1 and τ_2 form a deadlock, and τ_1 enter critical section $z_{1,i}$ which blocks τ_2 before τ_2 enters critical section $z_{2,j}$ which blocks τ_1. Because critical section $z_{1,i}$ blocks τ_2, the read/write priority ceiling $RWPL_k$ of the data object O_k locked by τ_1 when τ_1 enters $z_{1,i}$ should be no lower than the priority of τ_2. In other words, the lock request of τ_2 to enter critical section $z_{2,j}$ should not succeed until τ_1 leaves $z_{1,i}$, and no deadlock should occur. $\Box$

Theorem 2 *The maximum number of priority inversion per transaction instance is one.*

Proof. Let a transaction instance τ_H be blocked by two distinct lower-priority transaction instances τ_L and τ'_L. Since there is no transitive blocking (please see Lemma 2), τ_L and τ'_L must be executing in critical sections $z_{L,i}$ and $z_{L',j}$ to directly block τ_H, respectively. Let τ_L enter critical section $z_{L,i}$ before τ'_L enters critical section $z_{L',j}$. Since critical section $z_{L,i}$ blocks τ_H, critical section $z_{L,i}$ should also block τ'_L because the priority of τ_H is higher than the priority of τ'_L. In other words, τ'_L should not enter critical section $z_{L',j}$ to directly block τ_H until τ_L leaves $z_{L,i}$, and the maximum number of priority inversion per transaction instance should not be more than one. $\Box$.

Note that when transactions do abort, Theorems 1 and 2 remain correct, provided that aborted transactions must unlock their data objects. This is because aborted transactions will not be in a deadlock cycle or introduce any further priority inversion.

We shall first prove Lemmas 3, 4, and 5 to show that all of the 2VPCP schedules comply with the compatibility matrix shown in Table 1:

Lemma 3 *When data object O_k is read-locked by a transaction instance τ, no transaction instance can certify-lock O_k under the 2VPCP protocol.*

Proof. The lemma can be proven by contradiction. Let a distinct transaction instance τ' receive a certify lock on O_k when O_k is read-locked by a transaction instance τ under the 2VPCP protocol. By the definitions of the 2VPCP protocol, the priority of τ' must be higher than $RWPL_k$ (i.e., WPL_k), where $RWPL_k$ is no less than the original priority of τ'. In other words, τ' must inherit the priority of some transaction instance which is higher than $RWPL_k$ to certify-lock O_k. Let z'_i be the earliest critical section which τ' enters and which later blocks some higher-priority transaction instance τ'' whose priority is higher than $RWPL_k$. Based on Lemmas 1 and 2, τ' must be executing in critical section z'_i before τ'' is initiated.

There are two cases for discussions on when τ' enters critical section z'_i: Suppose that data object O_k is read-locked by transaction instance τ before τ' enters critical section z'_i. Since the 2VPCP protocol should not allow τ' to enter critical section z'_i because the priority of τ' is no higher than $RWPL_k$, a contradiction exists. (We assume that τ' enters a critical section which blocks a transaction instance with a priority higher than $RWPL_k$ in the past paragraph.)

Let data object O_k be read-locked by transaction instance τ after τ' enters critical section z'_i. The priority of τ must be higher than the priority of τ''; otherwise, data object O_k cannot be read-locked by transaction instance τ. If the priority of τ is really higher than the priority of τ'', then τ' has no chance to regain the CPU and issues a certify lock on O_k unless τ is blocked (when τ' issues a certify lock on O_k). Since there is no deadlock and no transitive blocking (please see Lemma 2 and Theorem 1), τ must be blocked by τ'. Since τ' blocks τ, τ' must be executing in a critical section which later blocks τ before τ is initiated (please see Lemma 1). It contradicts the assumption that data object O_k is read-locked by τ because τ has no way to read-lock O_k. $\square$

Lemma 4 *When data object O_k is write-locked by a transaction instance τ, no transaction instance can write-lock or certify-lock O_k under the 2VPCP protocol.*

Proof. Since both read and write locks set $RWPL_k$ as WPL_k, and the sets of transactions which may issue write or certify locks are the same, this lemma can be proven in a way similar to the proof of Lemma 3. $\square$

Lemma 5 *When data object O_k is certify-locked by a transaction instance τ, no transaction instance can lock O_k in any way under the 2VPCP protocol.*

Proof. This lemma can be proven in a way similar to the proof of Lemma 3 (by replacing every occurrence of "read-lock by τ", "$RWPL_k$", and "certify-lock by τ'" with "certify-lock by τ", "APL_k", and "lock by τ'", respectively). □

Theorem 3 *All 2VPCP schedules satisfy the compatibility matrix shown in Table 1.*

Proof. The correctness of this theorem directly follows from Lemmas 3, 4, and 5. □

Theorem 4 *All 2VPCP schedules are serializable.*

Proof. Since schedules generated by the 2-version 2PL protocol is (one-copy) serializable (1SR) [4, 9], and the set of schedules generated by the 2VPCP protocol is a subset of that generated by the 2-version 2PL protocol (Please see Theorem 3), all 2VPCP schedules are serializable. Note that all schedules which satisfy the 2PL scheme and the compatibility matrix in Table 1 are 2-Version 2PL schedules [4, 9]. Since all 2VPCP schedules satisfy the 2PL scheme and the compatibility matrix in Table 1, all 2VPCP schedules are 2-Version 2PL schedules. □

3 Read-Only Transaction Processing

3.1 Overview

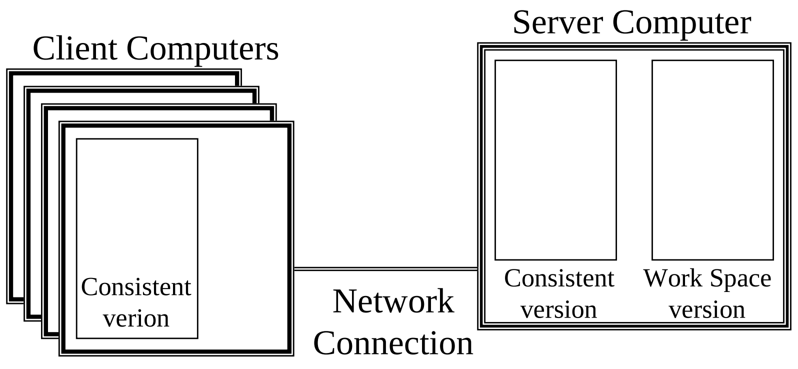


Figure 3: A client-server architecture for read-only transactions

The purpose of this section is to extend the 2VPCP protocol into local processing of read-only transactions, as shown in Figure 3. For the purpose of this paper, we assume that all updating transactions are submitted to the server-side system for execution. The main idea in this 2VPCP extension is to "duplicate" a consistent version of the database image at each client-side system to service read-only transactions locally at client-side systems. There are two major advantages of this approach: (1) A potentially large number of queries, i.e., read-only transactions, can be screened out of the normal operation of a real-time database system (at the server side). (2) Real-only transactions can be processed much faster and efficiently at client-side systems without going through potentially jammed network.

As astute readers may notice, higher-priority read-only transactions at the server system are already favored by the 2VPCP protocol because higher-priority read-only transactions can read-lock and access any data objects, unless the consistent versions of the data objects are under modifications (i.e., locked in a certify mode). We surmise that, in normal operation, the interval of a certify lock should not be long for server-side transactions because usually only a committing transaction tries to obtain a certify lock. The real question here is how to improve the response time of lower-priority (and even higher-priority) read-only transactions issued by users at client-side systems. The main idea is to maintain a consistent database image at each client-side system to improve the response time of users' local lower-priority and higher-priority read-only transactions and, at the same time, without sacrificing the serializability correctness of the entire system. In order to achieve this goal, we must build a serializability order of all transactions executing at client-side and server-side systems.

The technical question here is how to efficiently maintain a consistent database image at each client-side system which satisfies the above condition. Our approach is to let each of the client-side systems autonomously fabricate the consistent version of the server-side two-version database at the client side, such that all executions of read-only transactions at the client side can be properly inserted into the serializability order of transaction executions at the server side. In order to maintain a consistent database image at each client-side system, each transaction (or the system) must send client systems a message similar to a redo log (τ_i, *object_name*, *old_value*, *new_value*) for each of its write operations (to the working version of the server-side database). Each of the client-side systems then maintains its consistent database image by observing these messages. Note that the client-side consistent database images will be used for efficient failure recovery in Section 4, and no processing of redo logs is needed again during failure recovery. *old_value* in each redo-log message can be removed because it will not be used in any way.

For the rest of this paper, we assume that all messages sent in the network arrive at the destination in their sending order.

3.2 The Serializability-Order Rebuilding Mechanism

Because of the existence of a two-version database and the "preemptions" of conflicting transactions (which result in the effects of dynamic adjustment of serializability order) at the server-side system, the serializability order of transaction executions at the server side cannot be simply observed by the timestamps of successful conflicting locking requests issued by the server-side transactions. We propose to observe the serializability order of server-side transactions based on the order of the beginning of the shrinking phase of the server-side transaction instances. The beginning of the shrinking phase of any server-side transaction instance can be easily observed by the appearance of the first unlock request of the transaction instance. The information is purely syntactic and can be easily observed by the system with very low overheads because all server-side transaction instances issue lock and unlock requests to the system, regardless of whether the system tries to observe the beginning of their shrinking phase.

Let *begin_unlock* denote the first unlocking operation of a transaction instance. We shall prove in the following theorem that the begin_unlock order of transaction instances at the server side complies with the serializability order of the transaction instances executing at the server side. This observation provides a simple and efficient mechanism (which we will show you in the next section) to determine the serializability order of server-side transactions.

Theorem 5 *The begin_unlock order of transaction instances (at the server side) complies with the serializability order of the transaction instances (at the server side).*

Proof. This theorem can be proven by considering all of the combinations of conflicting r/w operations. To determine the serializability order of conflicting transaction instances, four cases must be considered, as shown in Figure 4, where $WL_i(x)$, $RL_i(x)$, $CerL_i(x)$, and $Begin\_Unlock$ denote the write lock, read lock, certify lock, and begin_unlock message of transaction τ_i. Note that the 2VPCP protocol satisfies the 2PL scheme and the compatibility matrix shown in Table 1 (Please see Theorem 3 and the definitions of the 2VPCP protocol).

1. If there is a write/write conflict between two conflicting transaction instances, e.g., τ_1 and τ_2 in Figure 4.a, then the begin_unlock order of transaction instances must comply with the serializability order of the transaction instances. This is because write and certify locks are incompatible with each another. Every write lock of a transaction instance must precede its begin_unlock.

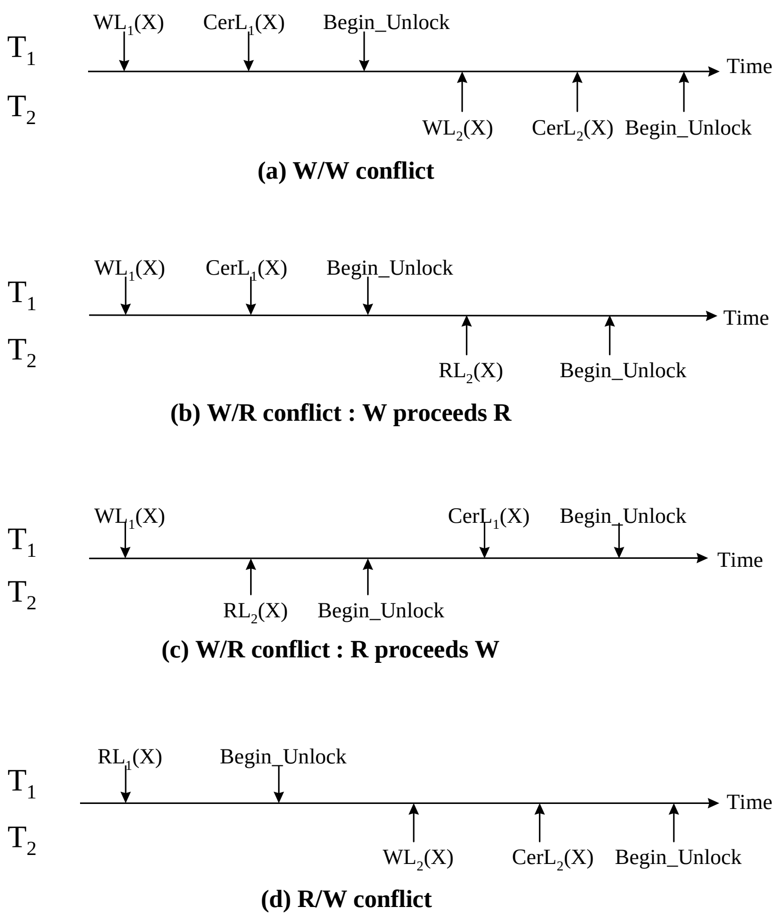


Figure 4: Serializability order of conflicting transactions

2. If there is a write/read conflict between two conflicting transaction instances, and the read lock of a transaction instance is granted after the write lock and certify lock of another transaction instance, e.g., τ_1 and τ_2 in Figure 4.b, then the begin_unlock order of transaction instances must comply with the serializability order of the transaction instances. This is because read and certify locks are incompatible. The unlocking of the certify lock must be earlier than the granting of the read lock.

3. If there is a write/read conflict between two conflicting transaction instances, and the read lock of a transaction instance is granted before the certify lock, but after the write lock of another transaction instance, e.g., τ_1 and τ_2 in Figure 4.c, then the begin_unlock order of transaction instances must comply with the serializability order

of the transaction instances. This is because read and certify locks are incompatible. The certify lock cannot be granted until the unlocking of the read lock. Obviously, the fact that τ_2 may have accessed the consistent version lets τ_2 be before τ_1 in the serializability order.

4. Suppose that there is a read/write conflict between two conflicting transaction instances, and the read lock of a transaction instance precede the write lock of another transaction instance, e.g., τ_1 and τ_2 in Figure 4.d. Regardless of what the order of the begin_unlock of τ_1 and the write lock $WL_2(x)$ of τ_2 is, the begin_unlock order of transaction instances τ_1 and τ_2 must comply with the serializability order of the transaction instances. This is because the transformation of the write lock into the certify lock must be blocked by the read lock if the read lock is not released. Note that τ_1 may release a lock after τ_2 obtains a conflicting write lock when the begin_unlock of τ_1 is after the write lock $WL_2(x)$ of τ_2.

$\square$

Note that if the server-side system allows conflicting transaction instances to commit in an order different from their serializability order, then each client-side system must apply the redo logs of the committing transaction instance on the consistent database version of the system in order of their begin_unlock messages, instead of their commit order. Theorem 5 provides the general relationship between the serializability order of server-side transaction instances and their order of begin_unlock messages, regardless of whether the server-side system may or may not crash. However, we must emphasize that if the server-side system indeed allows conflicting transaction instances to commit in an order different from their serializability order, and the server may crash at any time, then some schedules of server-side transactions are not recoverable according to the definition of recoverable schedules [9]. In the next section, we shall address this recovery issue for the extended 2VPCP protocol at the server side.

3.3 The Extended 2VPCP Protocol

We now present the mechanism in extending the 2VPCP protocol for local processing of read-only transaction instances at client sides. We assume that messages sent in a network arrive at destination systems in the order of their sending times, and the network is reliable. We are interested in a close environment, where the network is close and under reasonable control. A network protocol such as TCP/IP which supports reliable message transmissions is adopted. If the network fails, then the extended 2VPCP protocol, similar to many other distributed concurrency control protocols, will not work in a distributed environment.

Server-Side Transactions:

Each server-side transaction instance τ_i (scheduled by the 2VPCP protocol) is required to broadcast a message to all client-side systems under the following three circumstances: Note that the server system can broadcast the messages on behalf of the server-side transaction instances.

1. Before τ_i writes on the working version of a data object O_j, τ_i (or the server-side system) must broadcast a message similar to a redo log (τ_i, O_j, old_value, new_value) to all client-side systems.
2. Before τ_i commits, τ_i (or the system) must broadcast a message similar to a commit log $(\tau_i, commit)$ to all client-side systems.
3. When τ_i first unlocks any data object, τ_i (or the server-side system) must broadcast a message $(\tau_i, begin\_unlock)$ to all client-side systems to signal the beginning of the shrinking period of τ_i.

Note that if the server system allows conflicting transaction instances to commit in an order different from their serializability order, and the server may crash at any time, then some schedules of server-side transactions are not recoverable according to the definition of recoverable schedules [9]. In order to maintain the recoverability of the system, the server system must delay the commitment of a transaction instance (i.e., the actual releasing of certify locks and the sending of commit log to client-side systems) until all preceding transaction instances in the serializability order (i.e., their order of begin_unlock messages) commit. However, if the server system may never crash, the above delaying requirement of commitment is not necessary. In other words, the above delaying requirement is not necessary before Section 4 which is for failure recovery.

Client-Side System:

Let each client-side system and the server-side system share the same consistent version of the database initially. A transaction instance is said to *have committed at a client-side system* if all of the redo logs of the transaction instance have been applied on the consistent database version of the client-side system. Note that the write through procedure of each committing transaction instance at the server-side computer cannot be completed until all redo logs and commit log are delivered to all client-side systems. When a reliable broadcasting network is adopted, the server-side system may simply return from any sending operations and assume that all client-side systems will receive the messages sent by these operations eventually. The details regarding logging and commitment of server-side transaction instances will be discussed in Section 4.2.

During the system operation, each client-side system keeps all redo-log messages of server-side transaction instances which have not committed at the client-side system. When a client-side system receives a commit message, the system applies the redo logs of the committing transaction instance on the consistent database version of the system atomically in order of their begin_unlock messages. Note that the client-side systems may be busy doing something else; so there could be a difference between copies on the client sides and on the server side. However, sooner or later, the client-side systems will catch up when the local workloads drop. Such a phenomenon will not cause any problem because all 2VPCP schedules with local read-only transaction processing are serializable even though some clients might actually apply the logs later (Please see Theorem 6).

Each client-side system has a unique updating transaction τ_{Upd} which is responsible for atomically updating the consistent database image based on redo-log messages of committing transaction instances. Before transaction τ_{Upd} updates the consistent database image, it must write-lock the entire database image. Transaction τ_{Upd} may be periodic or aperiodic. The higher the priority of τ_{Upd}, the faster τ_{Upd} can update the consistent database image by processing the redo logs to reflect the database image updated by committed server-side transactions. The assignment of a high priority to τ_{Upd} can also help in reducing the recovery time because the recovery mechanism must reflect the consistent database image updated by all of the committed server-side transactions, and the mechanism depends on τ_{Upd} to process the redo logs. However, the high priority of τ_{Upd} may interfere with the executions of read-only transactions at the client side. Simulation experiments regarding the priority of τ_{Upd} will be included in Section 5.

Autonomous Read-Only Transaction Processing at Client-Side Systems:

Each client-side system should schedule all of its transaction instances including τ_{Upd} in a preemptive priority-driven fashion. Before a read-only transaction instance reads any data object, it must read-lock the entire database image. Note that the entire database can be locked or unlocked by simply locking or unlocking a global flag. (Simulation experiments will be done to justify the setting of the priority of τ_{Upd}.)

Note that the consistent database image at the server-side system may not be the same as the consistent database image at some client-side systems. It is even possible that the τ_{Upd} transactions of different client-side systems process the redo logs of committed transaction instances at different speeds, due to different workloads at different client-side systems. However, as shown in Theorem 6, all 2VPCP schedules with local read-only transaction processing are serializable. Read-only transactions at different client-side systems may have consistent, but different views of the database.

Lemma 6 *All client-side systems have the same consistent database image if they receive the same set of messages sent from the server-side system and apply them to the database image.*

Proof. The correctness of this lemma follows directly from the assumption that the network delivers messages in a first-come-first-serve fashion. □

Lemma 7 *The database image maintained at client-side systems always satisfies the serializability order of server-side transaction instances.*

Proof. The correctness of this proof follows directly from Theorem 5 and the definitions of the database image maintenance mechanism. □

Theorem 6 *All 2VPCP schedules with local read-only transaction processing are serializable.*

Proof. Theorem 4 shows that all transactions at the server side are serializable. The problem is whether all client-side read-only transactions and all server-side transactions together are still serializable. Lemma 6 shows that all client-side systems have the same consistent database image if they receive the same set of messages sent from the server-side system and apply them to the database image. That is, there is no inconsistent view of the database among client-side transactions running on different client sides. Furthermore, Lemma 7 shows that the database image maintained at client-side systems always satisfies the serializability order of server-side transactions. A client-side read-only transaction is considered to occur exactly after the server-side transactions which commit at the corresponding client-side system before the read-only transaction read-lock and read the consistent database image. We conclude that client-side read-only transactions and all server-side transactions together are serializable. □

4 Failure Recovery

4.1 Motivation

The purpose of this section is to further extend the 2VPCP protocol to the failure recovery of the server-side system. As astute readers may notice, the 2VPCP protocol can be applied in both memory or disk resident databases. For the rest of this section, we will focus our discussions on memory-resident databases. We shall also require that the 2VPCP protocol only allows conflicting server-side transactions to commit in their serializability order. The enforcement of commit order can be done easily by delaying the commitment of transactions. However, the delaying of the commitment of server-side transactions may increase the maximum number of priority inversions for a server-side transaction by one. It can be explained by the following example:

Let a higher-priority transaction instance τ_H be blocked by a lower-priority transaction instance τ_L under the 2VPCP protocol at the server side. The *begin_unlock* message of τ_L will be before that of τ_H because of the adoption of the 2PL scheme. Since τ_H has a higher priority than τ_L does, τ_H may preempt τ_L (after τ_L unlocks the data object which blocks τ_H) and try to commit before τ_L commits. In order to let conflicting transaction instances commit according to their serializability order, i.e., the *begin_unlock* message order, τ_H will be delayed to wait for τ_L to commit to keep the system recoverable. Because there is no transitive blocking, and the maximum of priority inversions for the 2VPCP protocol is one, the extra number of priority inversions, due to the delaying of commitment, for τ_H is one.

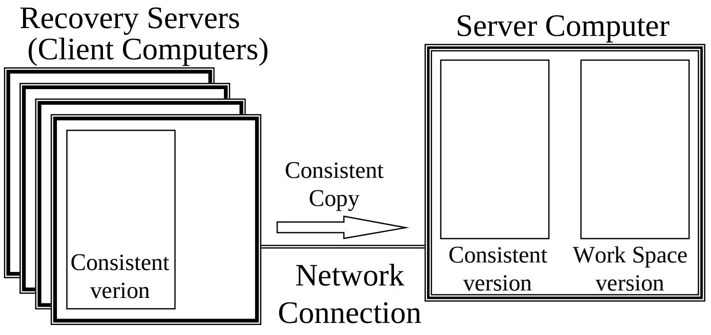


Figure 5: A client-server architecture for failure recovery

The client computers adopted for local processing of read-only transactions (in Section 3) are used as recovery servers in this section when the server-side system crashes. Since each client-side system maintains a consistent database image, the recovery procedure of the server-side system can be simply done by copying a client-side consistent database image to the server-side system. Note that there is no need for this recovery procedure to process any logs, and the time required to copy a large amount of data over a dedicated network, e.g., Fast Ethernet or ATM network, is not much more than the time to read the same amount of data from disks.

There are two major advantages to this approach: (1) The recovery procedure can be done very efficiently and predictably due to no processing of logs and no disk access. (2) The write-through procedure of a committing transaction instance at the server-side system can be done very efficiently and predictably because an unpredictable mechanical storage device, such as a disk, is replaced with an electronic transmission device (to transmit committing-related logs to client-side systems over network).

We must emphasize that the validity of the recovery mechanism proposed in this section relies on the assumption that all of the client computers and the server computer will not fail at the same time. Although the chance can be almost neglected when the number of client computers is reasonably large, the durability of transactions can not be maintained if all of the client computers and the server computer, indeed, fail at the same time. One alternative is to let at least one of the client computers write the consistent database image into stable storage. When all of the client computers and the server computer fail simultaneously, the consistent database image is restored from the stable storage. However, we must point out that the performance results described in the following sections will be significantly affected, at least on the client computer which is responsible for writing the consistent database image into stable storage, and the recovery time of the entire system will be increased by the amount of time in reading the consistent database image from the stable storage if all of the client computers and the server computer fail at the same time.

4.2 Logging and Recovery Mechanisms - A Server-Based Approach

Each server-side transaction instance τ_i must broadcast a redo log (τ_i, O_j, *old_value*, *new_value*) message to all client-side systems over a network, as required by the extended 2VPCP protocol in local processing of read-only transactions. The write through procedure of each committing transaction instance at the server-side computer cannot be completed until all redo logs and commit log are delivered to client-side network. When a reliable broadcasting network is adopted, the server-side system may simply return from any sending operations and assume that all client-side systems will receive the messages sent by these operations eventually. Note that the seek time and latency delay of a disk operation do, in general, average around $8.5ms$ and $4ms$, respectively, while the time in sending a $100B$ message over a Fast Ethernet network is only around $8us$. There is no collision problem in this architecture because all network traffics are either from server side (during normal operation) or from client side (during failure recovery).

When the server-side system crashes, any client-side consistent database image can be used to restart the server-side system. When network delay is considered, the consistent database image of the client-side system closest to the server-side system is selected for data copying, because the client-side system must receive all of the messages sent from the server-side system within a reasonable network delay which should be very small and predictable in a dedicated local-area network. Note that the recovery procedure of the server-side system will need to borrow the database image of any of the client-side systems for recovery. As a result, an additional workload will occur at the client-side system. Such copying of the database image will also increase the workload of the client-side system which lends the database image to the failed server-side system. Depending on how fast we want to recover the failed server-side system, the miss ratio of transactions running on the client-side system will be affected in a different degree. However, in this case, other client-side systems can still operate independently. When a client-side system fails, the server-side system and other client-side systems will not be affected because the failed client-side system operates independently from others. However, the recovery procedure of the failed client-side system will need to borrow the database image from another client-side system or the server-side system to re-build its database image. Such copying of the database image will also increase the workload of the system which lends the database image to the failed client-side system. Depending on how fast we want to recover the failed system, the miss ratio of transactions will be affected in a different degree.

4.3 System Overheads

The redo and commit logs sent by the server-side system (for local processing of client-side read-only transactions) are processed autonomously at each client-side system to produce a consistent database image without consuming any computation power from the server-side system. There is also no need to run any time-consuming checkpoint procedure at the server-side system. Furthermore, the logging procedure and the write-through procedure in the proposed approach can be done very efficiently and predictably by message sending. Note that memory-resident databases are very vulnerable to system failure. When compared to

\	Disk	RAM-Disk	Server
Checkpoint	needed	needed	no need
Log Processing during Recovery	needed	needed	no need
Write Through (100B)	12.5ms	31.49us	8.24us
Run-Time Logging (100B)	12.5ms	31.49us	8.24us

Table 2: The comparisons among disk-based, RAM-disk-based, and network-server-based recovery mechanisms.

the (mechanical) disk operations for logging and write-through procedures, the time spent in sending redo and commit log messages is significantly smaller.

Table 2 provides some performance comparisons among disk-based, RAM-disk-based, and network-server-based recovery mechanisms. $100B$ is assumed as the size of a log. Fast Ethernet is used for transmitting messages between two computers. Intel Value-Series-100 flash memory cards, which offer high-performance disk emulation, are used to estimate the performance of RAM-disks, where 32 bytes per word is assumed, and every word write and byte read cost $10us$ and $100ns$, respectively [3]. NEC 16-Megabit Rambus DRAM, which is an extremely high speed CMOS DRAM, is used to estimate the memory access time, where $40ns$ is needed for the first byte read; $2ns$ per byte thereafter [1]. IBM Ultrastar 2XP high-performance disk drives with SCSI-3 Wide are used to estimate the seek time and rotational latency of an access, the average sustained data rate is $9.2MB$ per second [2]. Let each word consist of 64 bits, and memory-remapping be used for copying parameters between the layers of the operating system. Suppose that the time to write $100B$ to a disk consists of the time for one disk I/O and the time for the memory copying of $100B$ log data (because of the memory copy for a system call). The time to write $100B$ to a RAM-disk consists of the time for writing $100B$ on the flash memory cards and the time for the memory copying of $100B$ log data (because of the memory copy for a system call). The time to transmit a $100B$-log message to a remote server consists of the time for transmitting $100B$ over the network, the time for the memory copying of $100B$ log data at the server (for making a send system call), and the time for the memory copying of $100B$ log data at the client (for receiving the message from the buffer).

The recovery times for disk-based, RAM-disk-based, and network-server-based recov-

\	Disk	RAM-Disk	Server
Recovery Time	$11.07 sec$ + log processing time	$10.2 sec$ + log processing time	$8.4 sec$

Table 3: The comparisons among the recovery times for disk-based, RAM-disk-based, and network-server-based recovery mechanisms.

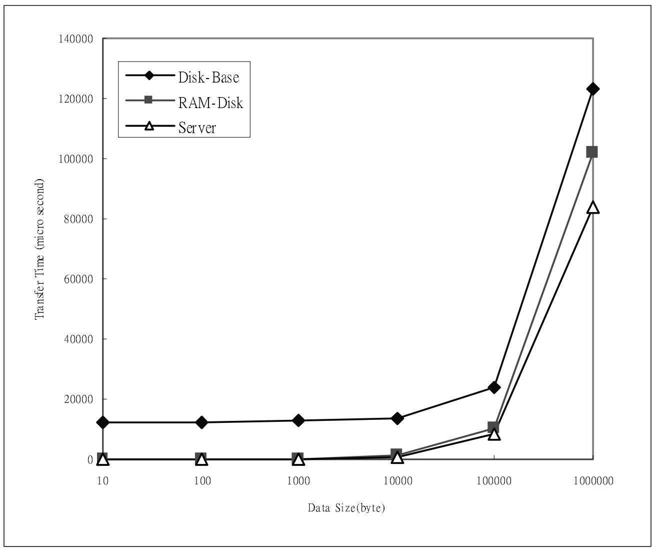


Figure 6: Data access time plus memory-copying time for databases of various sizes: no log processing time is included.

ery mechanisms are estimated as follows: Let a real-time database consist of $100MB$ data. The recovery time for the network-server-based recovery mechanism consists of the time for transmitting $100MB$ over the network, the time for the memory copying of $100MB$ data at the server, and the time for the memory copying of $100MB$ data at the client (for sending the image). The recovery time for the disk-based recovery mechanism consists of the time for reading $100MB$ from the disks, the time for the memory copying of $100MB$ data (from the buffers or controller), and the time for processing logs. The recovery time for the RAM-based recovery mechanism consists of the time for reading $100MB$ from the flash memory cards, the time for the memory copying of $100MB$ data (from the buffers or "controller"), and the time for processing logs.

Figure 6 shows the data access time plus memory-copying time for databases of various sizes. The network-server-based recovery mechanism consumes less time for data access and memory-copying than the other approaches do. We should also point out that, if a high-bandwidth ATM network is used, the recovery time for the network-server-based recovery mechanism on a $100MB$ database may drop to $1.6sec$. The log processing time for disk and RAM-disk approaches, which involves more reads from disks or RAM-disks, may further degrade their performance. Because of the saving in run-time checkpointing and recovery-time log-processing, we surmise that the proposed network-server-based approach should have outstanding performance, compared to other approaches. The major drawback of this approach is the hardware cost. However, we should emphasize that such cost is for both processing of read-only transactions and failure recovery, and much run-time performance can be obtained.

5 Performance Evaluation

5.1 Data Sets and Measurement

The experiments described in this section are meant to assess the capability of the 2VPCP protocol in transaction processing, especially when a large number of read-only transactions execute at client sides. Our simulation experiments compare the performance of the Two-Version Priority Ceiling Protocol (2VPCP), Priority Ceiling Protocol (PCP) [29], the Read/Write Priority Ceiling Protocol (RWPCP) [30], the new Priority Ceiling Protocol (NPCP) [17], and the two-version two-phase locking protocol (2V2PL) [4, 9] in both single and multiple processor environments.

The primary performance metric used is the miss ratio of a transaction, referred to as *Miss Ratio*. The *Miss Ratio* of each transaction τ_i is the percentage of requests of transaction τ_i that miss their deadlines. Let num_i and $miss_i$ be the total number of transaction requests and deadline violations during an experiment, respectively. *Miss Ratio* is calculated as $\frac{miss_i}{num_i}$.

The test data sets were generated by a random number generator. The number of transactions per transaction set was randomly chosen between 10 and 30, where a transaction set is a set of transactions. The experiments started with transaction sets of an utilization factor equal to 60% and increased by 5% at a time until the utilization factor was equal to 95%, where the utilization factor of a transaction set is equal to the sum of the ratio of the computation requirements and the period of every transaction in the set.

parameter	value
transaction number per set	$(10, 30)$
utilization factors	$(60\%, 95\%)$
transaction period	$(10, 10000)$
the number of data objects read or written by a transaction	$(1, 10)$ for ROT $(1, 5)$ for the rest
processor number	$1, 2, 4, 6$ or 11
simulation time	$1,000,000$

Table 4: Parameters of simulation experiments, where ROT stands for read-only transactions.

Each transaction set was simulated from time 0 to time $1,000,000$. Over 100 transaction sets per utilization factor were tested in the uniprocessor environment, and their results were averaged. 25 transaction sets per utilization factor were tested in the two-processor, 4-processor, 6-processor, and 11-processor environments, and their results were averaged. The period of a transaction was randomly chosen in the range $(10, 10000)$. The priority assignment of transactions follows the rate monotonic priority assignment [23]. The utilization factor of each transaction was no larger than 30% of the total utilization factor of the transaction set. The numbers of data objects read and written by an (update) transaction was both between 1 and 5. The number of data objects read by a read-only transaction was between 1 and 10. The critical sections of a transaction in locking data objects were evenly distributed and properly nested for the duration of the transaction. (This paper aims at critical real-time systems such as the avionics example or satellite control systems which have less than hundreds of data objects in the system, and the number of data objects which are read and written is, in general, not large.) The parameters are summarized in Table 4.

When a single processor environment was simulated, ordinary transaction sets (without read-only transactions) were simulated. When a multiple processor environment was simulated, one processor ran a transaction set consisting of update transactions, and the other processors ran transaction sets consisting of read-only transactions. The utilization factor on each processor was the same. The purpose of the multiprocessor simulations are meant to access the capability of the 2VPCP protocol in processing client-side transactions.

5.2 Experimental Results

5.2.1 Uniprocessor Environments

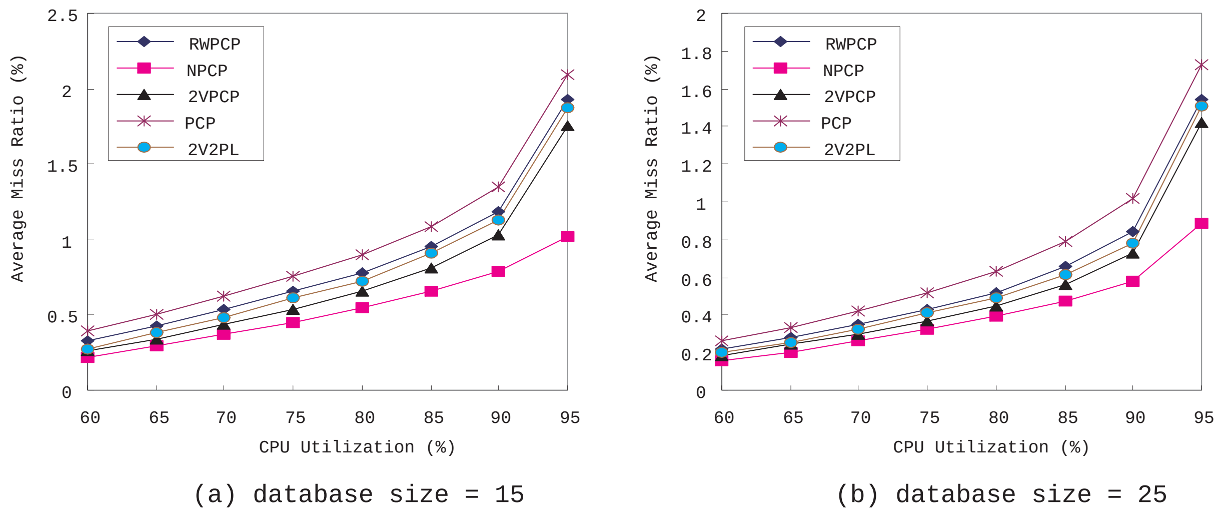
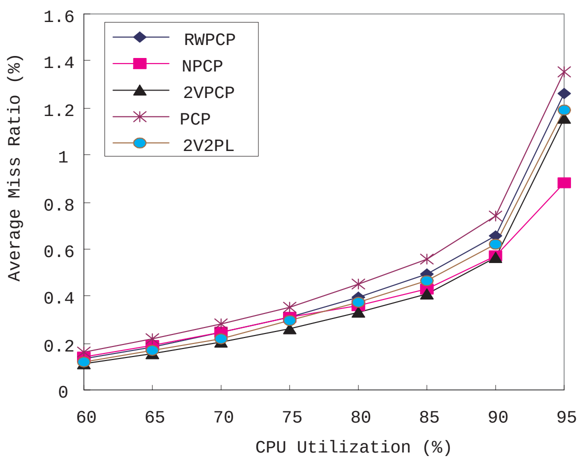
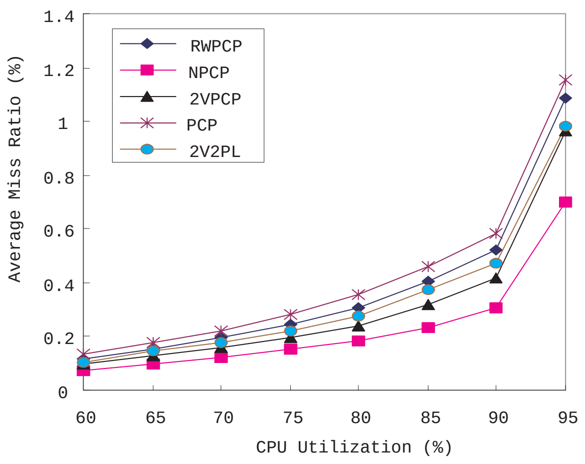


Figure 7: The miss ratio of all transactions: single processor, database size was 15 or 25.

Figures 7, 8, and 9 show the miss ratio of the entire transaction sets scheduled by the Priority Ceiling Protocol (PCP) [29], Read/Write Priority Ceiling Protocol (RWPCP) [30], new Priority Ceiling Protocol (NPCP) [17], the two-version two-phase locking protocol (2V2PL) [4, 9], and Two-Version Priority Ceiling Protocol (2VPCP), when the number of data objects in the system ranged from 15 to 200. NPCP outperformed 2VPCP and other protocols in uniprocessor environments. PCP which did not consider read and write semantics had the worst miss ratio among the others. The reason why NPCP outperformed others was because each transaction scheduled by NPCP adopted its own local copy for write operations. In other words, multiple versions of a data object might co-exist in a NPCP-scheduled system. On the other hand, 2VPCP only had two versions for each data object and was a compromise between NPCP and RWPCP, where there was only one version for each data object in RWPCP-scheduled systems. Here 2VPCP outperformed 2V2PL because of better priority inversion control. The experiments revealed the impacts of data versions on the miss ratio of different concurrency control protocols. We refer interested readers to Figures 10, 12, and 13 to compare the performance of different algorithms in multiprocessor environments and in meeting the deadlines of the top 1/4 highest-priority transactions. Also, as shown in Figures 7, 8, and 9, the larger the database size was, the



(a) database size = 50



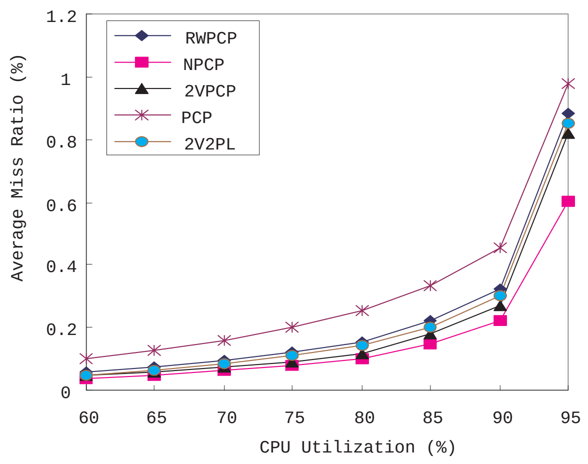
(b) database size = 100

Figure 8: The miss ratio of all transactions: single processor, database size was 50 or 100.

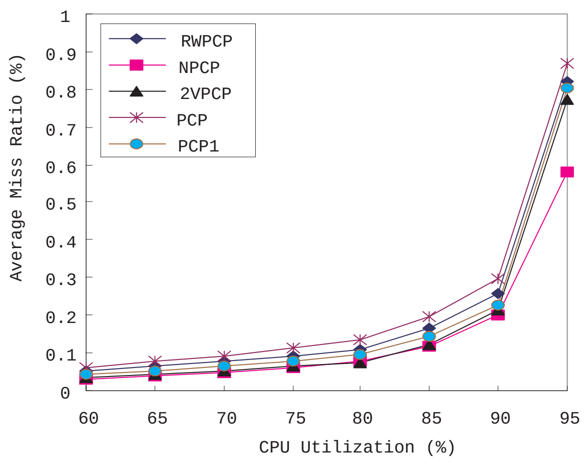
lower the miss ratio was, and the impacts of the number of data versions decreased. During the experiments, the maximum numbers of versions for a data object were 5.73, 4.91, 3.66, 3.18, 2.79, and 2.73 on average for a NPCP-scheduled system when the database size were 15, 25, 50, 100, 150, and 200, respectively.

Figure 10 show the miss ratio of the top 1/4 highest-priority transactions scheduled by PCP, RWPCP, NPCP, 2V2PL, and 2VPCP, when the number of data objects in the system was either 15 or 200. The miss ratio of the top 1/4 highest-priority transactions for a system with a database size between 15 and 200 was between their counterparts in Figure 10. We should point out that, when the number of data objects increased, the miss ratio of the top 1/4 highest-priority transactions scheduled by NPCP and 2VPCP almost collapsed. In other words, the schedulability of high-priority transactions scheduled by 2VPCP was greatly improved although there were only two versions per data object.

PCP, RWPCP, NPCP, 2V2PL, and 2VPCP outperformed the optimistic concurrency control protocol [22] because the optimistic concurrency control protocol restarted and aborted too many lower-priority transactions. On the other hand, the miss ratio of the top 1/4 highest-priority transactions scheduled by the optimistic concurrency control protocol was zero when the utilization factor of the system was no more than 100%, and the optimistic concurrency control protocol outperformed PCP, RWPCP, NPCP, 2V2PL, and 2VPCP in terms of the schedulability of high priority transactions.



(a) database size = 150



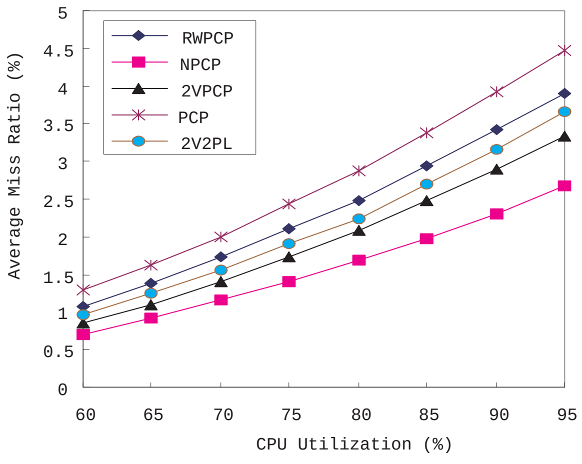
(b) database size = 200

Figure 9: The miss ratio of all transactions: single processor, database size was 150 or 200.

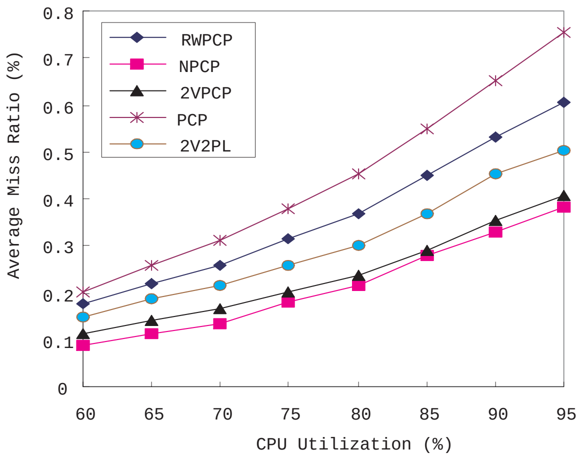
5.2.2 Multiprocessor Environments

Figures 12, 13, and 14 show the miss ratios of server-side transactions, client-side (read-only) transactions, and the top 1/4 highest-priority server-side transactions scheduled by PCP, RWPCP, NPCP, 2V2PL, and 2VPCP, when there are five client systems and one server system. The database size is either 50 or 100. Obviously 2VPCP greatly outperformed other protocols. The miss ratios of transactions scheduled by 2VPCP stayed very low because of the separation of concurrency control of server and client systems. The miss ratios of the top 1/4 highest-priority server-side transactions scheduled by 2VPCP was also substantially improved because of the same reason. Compared to uniprocessor environments, 2V2PL outperformed NPCP in multiprocessor environments because the two-version data concept greatly reduced the probability of access conflict.

Figures 15, 16, and 17 show the miss ratios of server-side transactions and client-side (read-only) transactions scheduled by PCP, RWPCP, NPCP, 2V2PL, and 2VPCP, when there were ten, three or one client systems. Obviously, 2VPCP performed consistently well, regardless of how many client systems existed. The performance of PCP, RWPCP, 2V2PL, and NPCP was very sensitive to the number of client systems. This was because a consistent version of the database image was maintained at each client-side system under 2VPCP to service read-only transactions locally at client-side systems. When PCP, RWPCP, 2V2PL, and NPCP were adopted, all transactions at the server and client sides competed for data objects. The larger the number of client systems, the larger the miss



(a) database size = 15



(b) database size = 200

Figure 10: The miss ratio of the top 1/4 highest-priority transactions: single processor, database size was 15 or 200.

ratios of the transactions.

Figures 18 and 19 shows the miss ratios of the top 1/4 highest-priority and total client-side transactions, respectively, when the updating transaction τ_{Upd} had the highest, top 1/4, middle, or lowest priority in the system. Note that τ_{Upd} updated the client-side consistent database image by processing the redo logs to reflect the database image updated by committed server-side transactions. The assignment of a high priority to τ_{Upd} could help in reducing the recovery time because the recovery mechanism had to reflect the consistent database image updated by all of the committed server-side transactions, and the mechanism depended on τ_{Upd} to process the redo logs. However, the higher the priority of τ_{Upd}, the more interference for the executions of other read-only transactions.

6 Conclusion

This paper proposes an integrated mechanism for real-time concurrency control and recovery. A two-version-based concurrency control protocol 2VPCP is proposed to reduce the blocking time of higher-priority transactions based on the idea of dynamic serializability adjustment [17, 19, 22] without relying on local data updates for transactions [17, 19, 22]. The 2VPCP protocol is, then, extended to a distributed environment to process read-only transactions at client-side systems locally and to support predictable and efficient failure

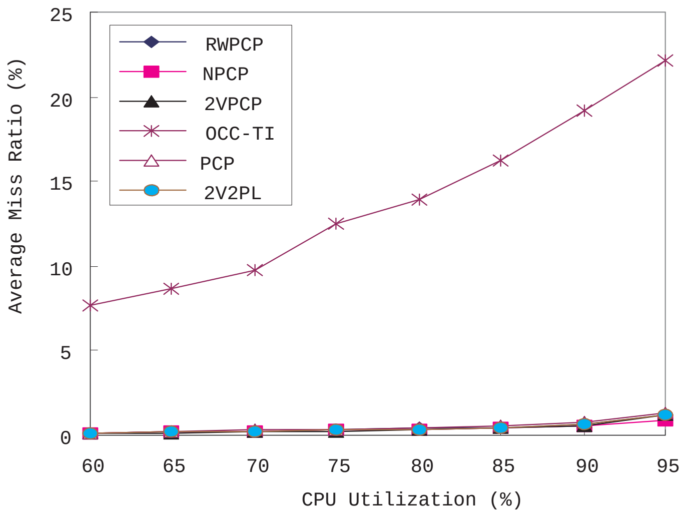
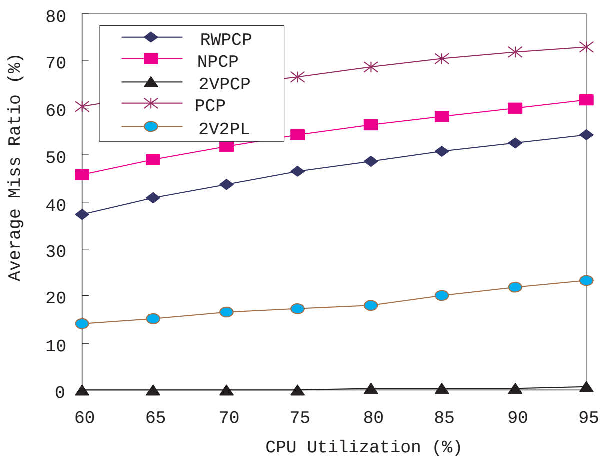


Figure 11: The miss ratio of all transactions scheduled by optimistic concurrency control protocol, PCP, RWPCP, NPCP, 2V2PL, and 2VPCP, when the database size was 50.

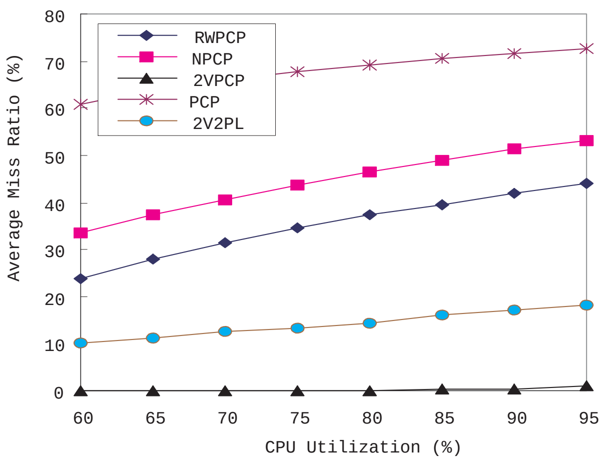
recovery. The performance of the algorithms was verified by a series of simulation experiments, for which we had very encouraging results. There are two major contributions in this paper: (1) The effectiveness of the two version approach is shown in reducing the blocking time of higher-priority transactions, and in improving the response time of client-side read-only transactions. (2) A two-version (network-server-based) architecture is proposed to not only support a disk-less run-time logging mechanism and an effective write-through procedure, but also provide an efficient and predictable recovery mechanism.

While issues in real-time concurrency control are better understood, the demand of system reliability is increasing. How to seamlessly integrate real-time concurrency control and recovery is becoming an important topic. For future research, we will extend the idea of two version databases into mobile environments, in which how to maintain a (partial) consistent copy at mobile clients is of paramount importance in improving the system performance. We will also extend the 2VPCP protocol into lower-level buffering control to further exploit recovery issues on disk-based databases.

Acknowledge: We thank Professor LihChyun Shu for his comments on this work.



(a) database size = 50



(b) database size = 100

Figure 12: The miss ratio of server-side transactions: one server, five clients, database size was 50 or 100.

References

[1] NEC 16M-bit Rambus DRAM, http://www.nec.com/necel/pdf/memory/488130L.PDF.

[2] IBM ultrastar 2xp 2xp 3.5-inch 4.5 and 9.1 GB high-performance disk drives, http://www.storage.ibm.com/oem/data/7059-95.html, 1996.

[3] Intel value series 100 flash memory card, http://developer.intel.com/design/fcard/prodbref/29768602.htm, 1998.

[4] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. In *ACM Transactions on Database Systems*, Jun 1980.

[5] A. Bestavros. Timeliness via speculation for real-time databases. In *the IEEE 15th Real-Time Systems Symposium*, 1994.

[6] L. Chih. Optimistic similarity-based concurrency control. In *the IEEE 1998 Real-Time Technology and Applications Symposium*, Jun 1998.

[7] L.B.C. Dipippo and V.F. Wolfe. Object-based semantic real-time concurrency control. In *the IEEE 14th Real-Time Systems Symposium*, Dec 1993.

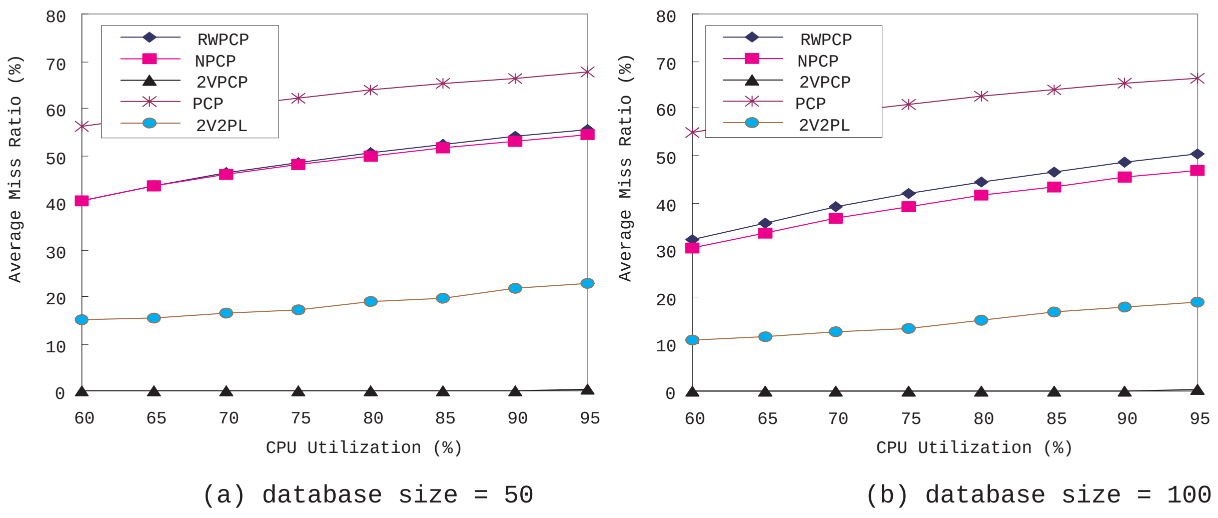


Figure 13: The miss ratio of client-side (read-only) transactions: one server, five clients, database size was 50 or 100.

[8] K. Elhardt and R. Bayer. A database cache for high performance and fast restart in database systems. In *ACM Transactions on Database Systems*, 1984.

[9] R.A. Elmasri and S.B. Navathe. In *Fundamentals of Database Systems*, 1994.

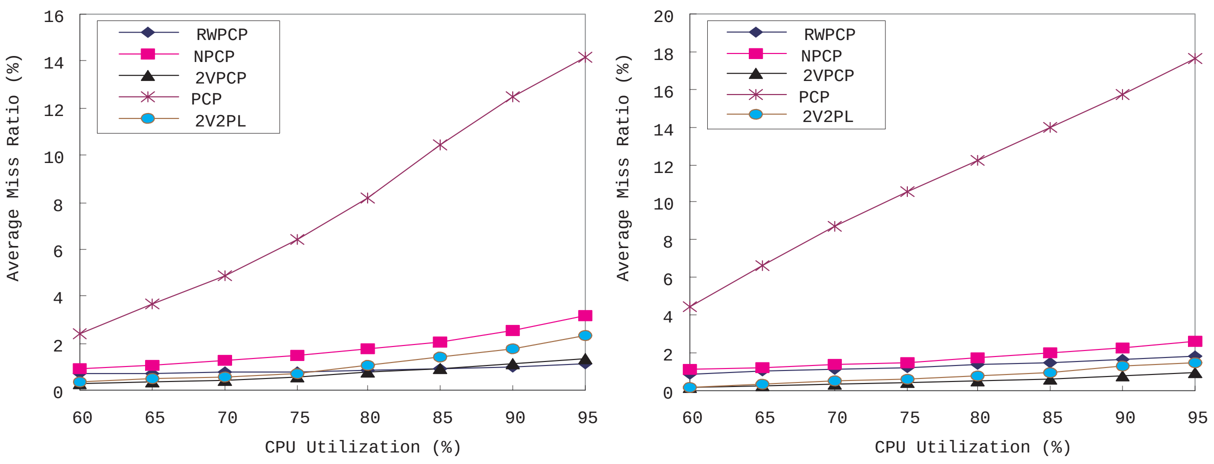
[10] R. Gupta, J. Haritsa, and K. Ramamritham. More optimism about real-time distributed commit processing. In *the IEEE 19th Real-Time systems Symposium*, Dec 1997.

[11] T. Haerder and A. Reuter. Principles of transaction oriented database recovery. In *ACM Computing Survey*, volume 15, December 1983.

[12] R. Hagmann. A crash recovery scheme for a memory resident database system. In *IEEE Transactions on Computers*, September 1986.

[13] J. Huang and L. Gruenwald. An update-frequency-valid-interval partition checkpoint technique for real-time main memory databases. In *the first International Workshop on Real-Time Databases: Issues and Applications*, 1996.

[14] M.U. Kamath and K. Ramamritham. Performance characteristics of epsilon serializability with hierarchical inconsistency bounds. In *the IEEE International Conference on Data Engineering*, April 1993.



(a) database size = 50 (b) database size = 100

Figure 14: The miss ratio of the top 1/4 highest-priority server-side transactions: one server, five clients, database size was 50 or 100.

[15] E. Kayan and O. Ulusoy. Real-time transaction management in mobile computing systems. In *the 6th International Conference on Database Systems for Advanced Applications*, April 1999.

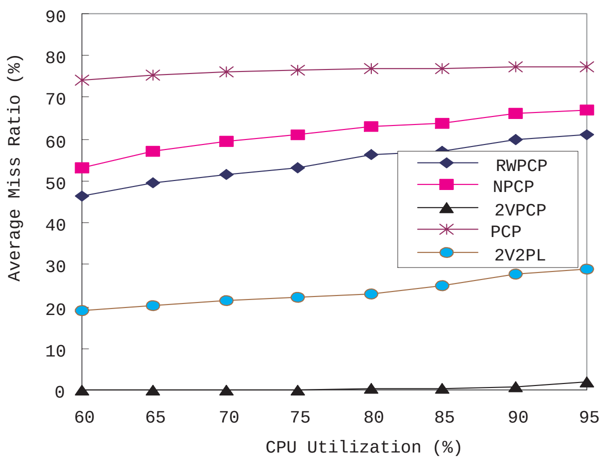
[16] T.-W. Kuo and A.K. Mok. SSP: a semantics-based protocol for real-time data access. In *the IEEE 14th Real-Time Systems Symposium*, December 1993.

[17] K-W. Lam and S.L. Hung. A preemptive transaction scheduling protocol for controlling priority inversion. In *the Third International Workshop on Real-Time Computing Systems and Applications*, Oct 1996.

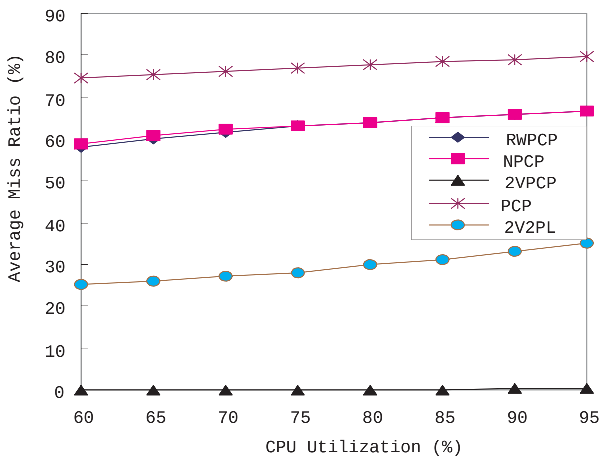
[18] K.-W. Lam, V. C.S. Lee, and S.L. Hung. Scheduling real-time read-only transactions. In *the Fourth International Workshop on Real-Time Computing Systems and Applications*, Oct 1997.

[19] K.-W. Lam, S.H. Son, V. C.S. Lee, and S.L. Hung. Using separate algorithms to process read-only transactions in real-time systems. In *the IEEE 19th Real-Time Systems Symposium*, December 1998.

[20] V. Lee, K. Lam, and T.-W. Kuo. Group consistency for real-only transactions in mobile environments. In *the 9th International Workshop on Parallel and Distributed Real-Time Systems*, April 2001.



(a) server transactions



(b) client transactions

Figure 15: The miss ratios of server-side transactions and client-side (read-only) transactions: one server, ten clients, database size was 50.

[21] V. C.S. Lee, K-W. Lam, and W. Tsang. Transaction processing in wireless distributed real-time databases. In *the 10th Euromicro Workshop on Real-Time Systems*, June 1998.

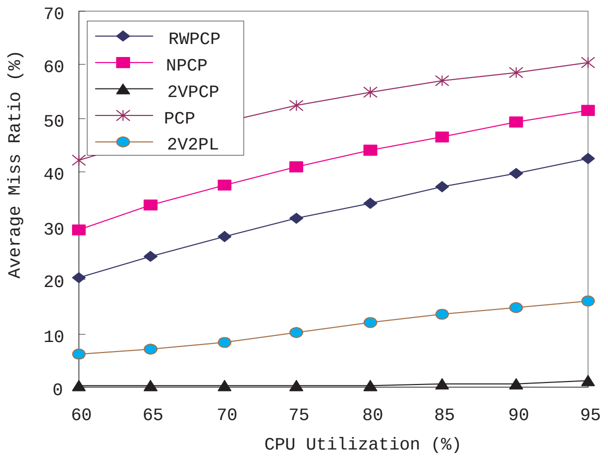
[22] Y. Lin and S. H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *the IEEE 11th Real-Time Systems Symposium*, December 1990.

[23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *JACM*, volume 20, January 1973.

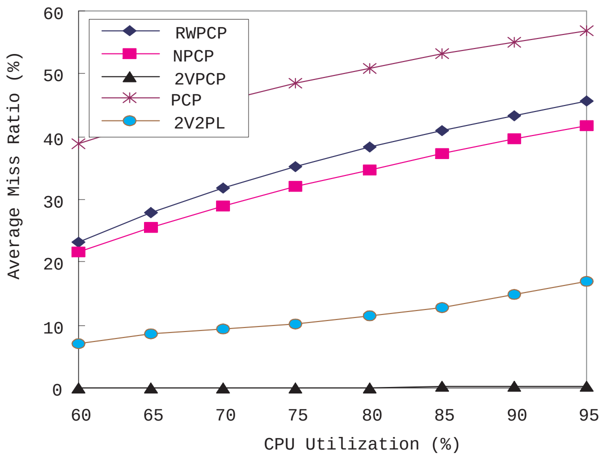
[24] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. In *ACM Transactions on Database Systems*, volume 17, March 1992.

[25] C.-S. Peng and K.-J. Lin. A semantic-based concurrency control protocol for real-time transactions. In *the IEEE 1996 Real-Time Technology and Applications Symposium*, 1996.

[26] C. Pu and A. Leff. Epsilon-serializability. In *Technical Report CUCS-054-90 Dept. of Computer Science, Columbia University*, January 1991.



(a) server transactions



(b) client transactions

Figure 16: The miss ratios of server-side transactions and client-side (read-only) transactions: one server, three clients, database size was 50.

[27] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. In *IEEE Transactions on Knowledge and Data Engineering*, volume 7, December 1995.

[28] L. Sha, R. Rajkumar, and J. L.P. Lehoczky. Concurrency control for distributed real-time databases. In *ACM SIGMOD Record*, volume 17, 1988.

[29] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Transactions on Computers*, volume 39, September 1987.

[30] L. Sha, R. Rajkumar, S. H. Son, and C.H. Chang. A real-time locking protocol. In *IEEE Transactions on Computers*, July 1991.

[31] R.M. Sivasankaran, K. Ramaritham, and J.A. Stankovic. Logging and recovery algorithms for real-time databases. In *Technical Report, University of Massachusetts*, 1997.

[32] M. Xiong, K. Ramamritham, R. Sivasankaran, J.A. Stankovic, and D. Towsley. Scheduling transactions with temporal constraints: Exploiting data semantics. In *the IEEE Real-Time Systems Symposium*, December 1996.

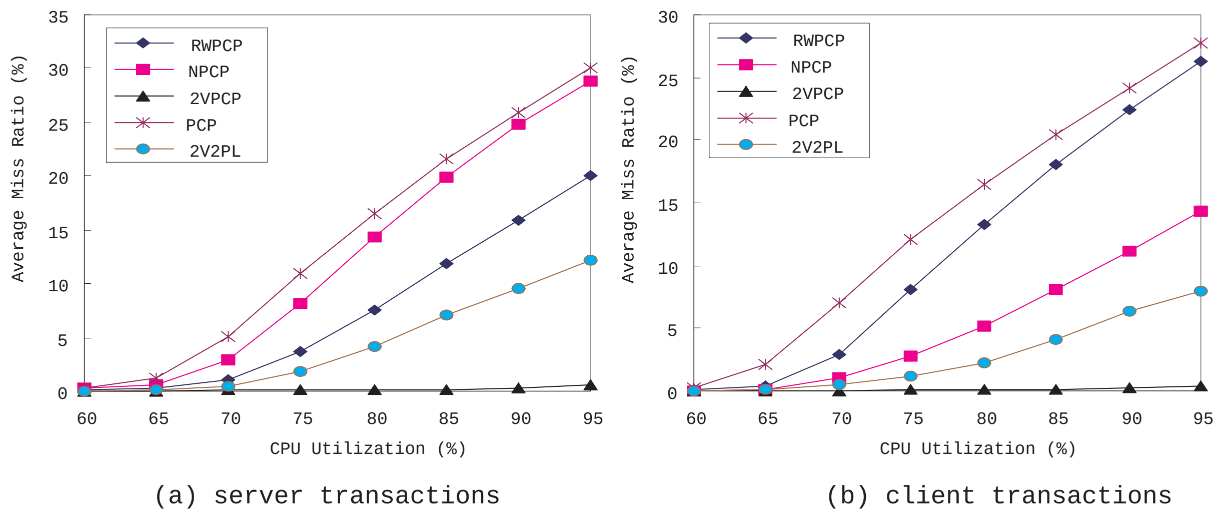


Figure 17: The miss ratios of server-side transactions and client-side (read-only) transactions: one server, one clients, database size was 50.

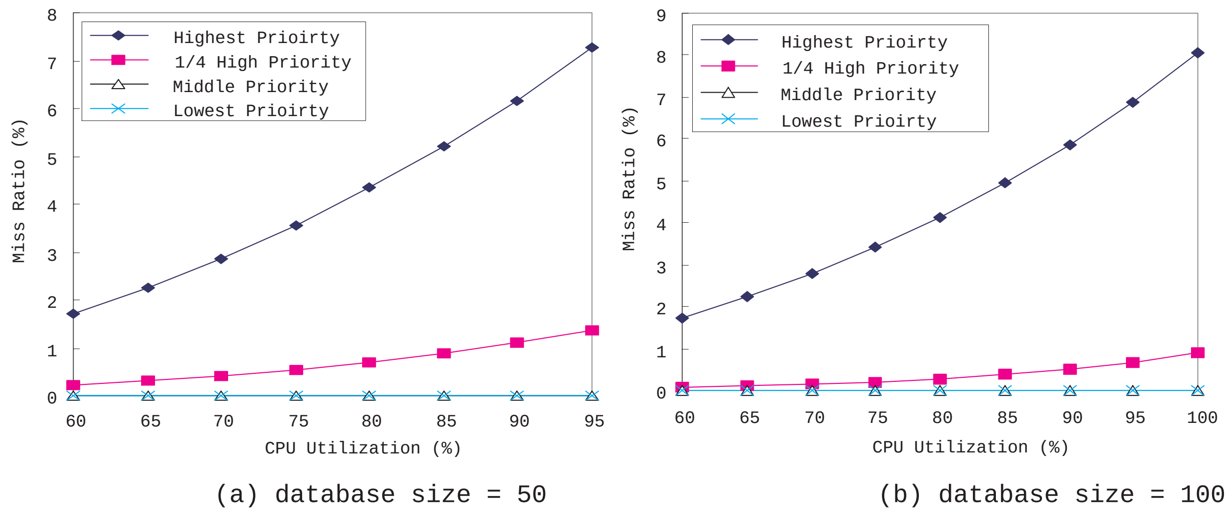
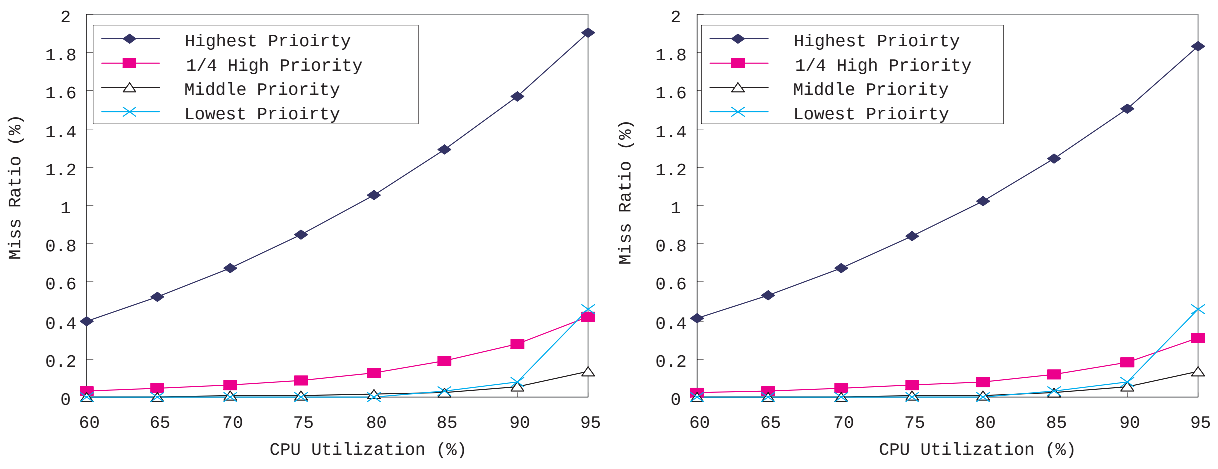


Figure 18: The miss ratio of the top 1/4 highest-priority client-side transactions: one server, five clients, database size is 50 or 100, the updating transaction τ_{Upd} had the highest, top 1/4, middle, or lowest priority in the system.



(a) database size = 50 (b) database size = 100

Figure 19: The miss ratio of the client-side transactions: one server, five clients, database size was 50 or 100, the updating transaction τ_{Upd} had the highest, top 1/4, middle, or lowest priority in the system.