Real-Time Concurrency Control in a Multiprocessor Environment

Tei-Wei Kuo, *Member*, *IEEE*, Jun Wu, and Hsin-Chia Hsih

Abstract—Although many high-performance computer systems are now multiprocessor-based, little work has been done in real-time concurrency control of transaction executions in a multiprocessor environment. Real-time concurrency control protocols designed for uniprocessor or distributed environments may not fit the needs of multiprocessor-based real-time database systems because of a lower concurrency degree of transaction executions and a larger number of priority inversions. This paper proposes the concept of priority cap to bound the maximum number of priority inversions in multiprocessor-based real-time database systems to meet transaction deadlines. We also explore the concept of two-version data to increase the system concurrency level and to explore the abundant computing resources of multiprocessor computer systems. The capability of the proposed methodology is evaluated in a multiprocessor real-time database system under different workloads, database sizes, and processor configurations. It is shown that the benefits of priority cap in reducing the blocking time of urgent transactions is far over the loss in committing less urgent transactions. The idea of two-version data also greatly improves the system performance because of a much higher concurrency degree in the system.

Index Terms—Real-time concurrency control, multiprocessor architecture, two-version database, priority inversion.

1 Introduction

Real-time concurrency control must maintain the consistency of the database but also meet the deadline requirements of real-time transactions. Although a number of researchers have proposed various real-time concurrency control protocols in the past decades, previous work mainly focuses on uniprocessor real-time databases (URTDB), e.g., [2], [13], [14], [15], [21], distributed real-time databases (DRTDB), e.g., [11], [25], or semantics-based concurrency control, e.g., [5], [7], [10], [19], [24]. Very little work has been done in exploring the abundant computing resources of multiprocessor computer systems.

With the advance of hardware and software technologies, there is an increasing demand for multiprocessor computing in real-time systems. How to better utilize the abundant computing resources of multiprocessor computer systems may significantly improve the performance or capacity of a real-time database system. The protocols designed for URTDB or DRTDB may not fit the needs of real-time database systems in a multiprocessor environment because of a potentially low concurrency degree and/or an increased number of priority inversions, where priority inversions is a situation in which a higher-priority transaction is blocked by a lower-priority transaction because of access conflict. A low concurrency degree in transaction execution may result in low system performance because protocols designed for URTDB or DRTDB do not try to utilize the extra computing power from multiple processors. On the other hand, the increase in the number of priority inversions is fatal to many critical real-time systems because the resulting long blocking time may fail critical timing constraints. The increase in the number of priority inversions is mainly caused by the fact that a lower-priority transaction which arrives later may block a higher-priority transaction running on another processor. Note that, when a uniprocessor real-time database is considered, no lower-priority transaction which arrives later may have a chance to lock any data object to block a higher-priority transaction because the CPU is always assigned to the ready transaction which has the highest priority in the uniprocessor system. We must emphasize that one of the main reasons to adopt multiple processors is to have enough computing power to meet the deadline requirements of transactions. It will be ironic if real-time transactions begin missing their deadlines because of a large number of priority inversions in a multiprocessor system.

The purpose of this research is to explore issues in concurrency control of multiprocessor-based real-time database systems. In particular, we are interested in techniques which can bound the number of priority inversions and, at the same time, better utilize the abundant computing power in multiprocessor environments. We shall show that the maximum number of priority inversions for transactions scheduled by uniprocessor protocols, such as *Read/Write Priority Ceiling Protocol (RWPCP)* [21], can be much more than one in a multiprocessor environment. In this paper, we will then propose the concept of priority cap to manage the priority inversions problem in a multiprocessor environment. We also explore the high concurrency of transaction executions by means of data versions without increasing the maximum number of priority inversions in the system. Each data object in the system

- *T.-W. Kuo is with the Department of Computer Science and Information Engineering National Taiwan University, Taipei, Taiwan 106, ROC. E-mail: ktw@csie.ntu.edu.tw.*
- *J. Wu and H.-C. Hsih are with the Department of Computer Science and Information Engineering National Chung Cheng University, Chiayi, Taiwan 621, ROC. E-mail: {junwu, ssc86}@cs.ccu.edu.tw.*

Manuscript received 5 June 2001; accepted 27 Nov. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 114299.

has two versions: The consistent version contains a data value updated by a committed transaction and the working version contains a data value updated by an uncommitted transaction. The capability of the proposed methodology is evaluated in a multiprocessor real-time database system under different workloads, database sizes, and processor configurations.

There are two major contributions in this paper: 1) We show that the number of priority inversions experienced by transactions scheduled by popular uniprocessor real-time concurrency control protocols, such as RWPCP, can be much more than one in a multiprocessor environment. This paper proposes a simple idea, called *priority cap*, to bound the maximum number of priority inversions in a multiprocessor-based real-time database system without much affecting the system performance. Note that a large number of priority inversions may result in deadline violations of critical transactions and is fatal to many time-critical systems. 2) We explore concurrent executions of conflicting read and write operations in multiprocessor environments, particularly with the idea of two-version data, to increase the concurrency of transaction executions. We must emphasize that the techniques proposed in this paper can be applied to many kinds of real-time applications which adopt multiple processors, such as defense application systems with data processing (e.g., HARM low-cost seeker), high-performance database systems, etc. Note that major database vendors all provide customers multiprocessor-based database systems. The techniques can also be applied to many other protocols. RWPCP is merely used to present the ideas proposed in this paper.

The rest of this paper is organized as follows: Section 2 summarizes the well-known RWPCP and illustrates the system model. Section 3 presents the motivation and the idea of priority cap. In particular, we show how to transform the well-known RWPCP into a protocol which guarantees one priority inversion for any real-time transaction. We then explore the concurrency of transaction executions by means of two-version database in multiprocessor environments. Section 4 addresses the implementation issues. Section 5 presents the simulation experiments of the proposed protocols in multiprocessor real-time database systems. Section 6 presents the conclusions.

2 Related Work

Although a number of researchers have proposed various real-time concurrency control protocols in the past decades, previous work mainly focuses on uniprocessor real-time databases (URTDB), e.g., [2], [13], [14], [15], [21], distributed real-time databases (DRTDB), e.g., [11], [25], or semantics-based concurrency control, e.g., [5], [7], [10], [19], [24]. Very little work has been done in exploring the abundant computing resources of multiprocessor computer systems. In particular, the Read/Write Priority Ceiling Protocol (RWPCP) [21] has shown the effectiveness of using read and write semantics in improving the performance of the Priority Ceiling Protocol (PCP) [20] in real-time concurrency control. Lam and Hung [13] further sharpened the RWPCP by proposing the idea of dynamic adjustment of serializability order for hard real-time transactions, where Lin and Son [15] first proposed the idea of dynamic adjustment of serializability order for optimistic real-time concurrency control. Kuo et al. [11] proposed using the idea of two-version databases to replace a delayed write procedure to save the extra space needed by the procedure. They proposed a two-version variation of the RWPCP [21] to have the flexibility in the dynamic adjustment of transaction serializability order favor higher-priority transactions and read-only transactions.

There has been very little work done for real-time transaction scheduling in multiprocessor environments. Lortz et al. [16] designed, implemented, and evaluated an object-oriented database system called MDARTS (Multiprocessor Database Architecture for Real-Time Systems). MDARTS ensures bounded locking delay by disabling preemption when a transaction is waiting for a lock and, hence, allows for the estimation of worst-case transaction execution times. Chiu et al. [6] compared the performance of the optimistic concurrency control (OCC) and the two-phase locking (2PL) in a multiprocessor real-time database system. A new locking protocol called 2PL-LW, which write-locks all data in the write set at once, was proposed. It was shown that OCC, in general, outperformed 2PL and 2PL-LW outperformed OCC. Bodlaender et al. [4] proposed a scheduling algorithm called Single-Queue Static-Locking strategy (SQSL) for a shared-memory, multiprocessor database system. Under SQSL, all transactions are handled in a first-come-first-served (FCFS) manner. Transactions are allowed to execute if there is no access conflicts among transactions. If there is any access conflicts, all transactions that arrive later must wait until no access conflict exists any more.

Different from any past work, this paper proposes a simple idea, called *priority cap*, to bound the maximum number of priority inversions in a multiprocessor-based real-time database system without much affecting the system performance. We also explore concurrent executions of conflicting read and write operations in multiprocessor environments, particularly with the idea of two-version data, to increase the concurrency of transaction executions. We must emphasize that the techniques proposed in this paper can be applied to many other protocols in the literature. RWPCP is merely used to present the ideas proposed in this paper.

3 System Model and Basic Uniprocessor Protocol

3.1 System Model

In this paper, we will explore the concurrency control of real-time database systems in a multiprocessor environment. The well-known uniprocessor protocol *Read/Write Priority Ceiling Protocol (RWPCP)* [21] will be used to present the ideas proposed in this paper.

Let a multiprocessor real-time database system consist of N processors and a number of data objects shared by transactions executing on different processors, where $N > 1$. We consider a multiprocessor real-time database system which consists of a fixed set of transactions executing in a multiprocessor real-time database and every transaction has a fixed priority.[1] Each transaction may issue a sequence of database operations: reads or writes. A transaction is a template of its instance and each instance will be instantiated for every request of the transaction. The allocation of transactions on each processor is done in an offline fashion and no transaction may migrate from one processor to another. The requests of a *periodic transaction* τ_i will arrive regularly for every period p_i. Each request of a transaction τ_i should be completed no later than its deadlines which is defined as its arrival time plus the relative deadline d_i of the transaction. If transaction τ_i is aperiodic, p_i is the minimal separation time between its consecutive requests. There are two kinds of locks in the system: read and write. Before a transaction reads (or writes) a data object, it must first read-lock (or write-lock) the data object. Data objects are shared by transactions executing on different processors. 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.
- 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 data object is earlier than the locking operation of another data object within a transaction instance, the corresponding unlocking operation of the former data object is later than the corresponding unlocking operation of the later data object. Note that it is one of the assumptions of RWPCP in handling the priority inversion problem.

3.2 Basic Uniprocessor Protocol

The well-known *Read/Write Priority Ceiling Protocol (RWPCP)* [21] is an extension of the Priority Ceiling Protocol (PCP) [20] in real-time concurrency control. While PCP only allows exclusive locks on data objects, RWPCP explores read and write semantics in improving the concurrency degree without sacrificing the schedulability of transactions. RWPCP is originally designed for a highly critical transaction system consisting of a fixed set of transactions running on a uniprocessor system, such as an avionics system [12] and a satellite control system [3] .

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 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 transactions. When a data object O_i is write-locked, the setting of $RWPL_i$ prevents any other transaction 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 with a sufficiently high priority to read-lock O_i in order to constrain the number of priority inversions for any transaction which may write-lock O_i because $RWPL_i$ is equal to WPL_i.

The well-known RWPCP was originally designed for scheduling transactions in a uniprocessor environment. The transaction which has the highest priority among all ready transactions (in a processor) is assigned the processor. The lock request of a transaction $\tau_{i,j}$ may be granted if its priority is higher than the highest read/write priority ceiling $RWPL_i$ of the data objects locked by other transactions. If not, 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 transactions 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 that locked O^*.

A transaction $\tau_{i,j}$ uses its assigned priority unless it locks some data objects and blocks higher priority transactions. If a transaction blocks a higher priority transaction, it inherits the highest priority of the transactions blocked by $\tau_{i,j}$. When a transaction unlocks a data object, it resumes the priority it had at the point of obtaining the lock on the data object. The priority inheritance is transitive. Note that the resetting of priority inheritance can be efficiently implemented by using a stack data structure.

It has been shown that no deadlock or transitive blocking is possible for RWPCP schedules in a uniprocessor environment and the maximum number of priority inversions for any transaction is one in a uniprocessor environment [21] . A transitive blocking is said to occur if a (higher-priority) transaction is directly blocked by another (lower priority) transaction which, in turn, is directly blocked by the other (further lower-priority) transaction. When all transactions follow a two-phase-locking (2PL) scheme where no transaction which adopts a 2PL scheme will request any lock after the transaction releases any lock, all RWPCP schedules are serializable.

4 Multiprocessor Protocols with Priority Caps

4.1 Motivation

With the advance of hardware and software technologies, there is an increasing demand for multiprocessor computing in real-time systems. How to better utilize the abundant computing resources of multiprocessor computer systems may significantly improve the performance or capacity of a real-time database system. The protocols designed for

1. They are the requirements of RWPCP. When the ideas proposed in this paper are applied to other protocols, these requirements may be relaxed.

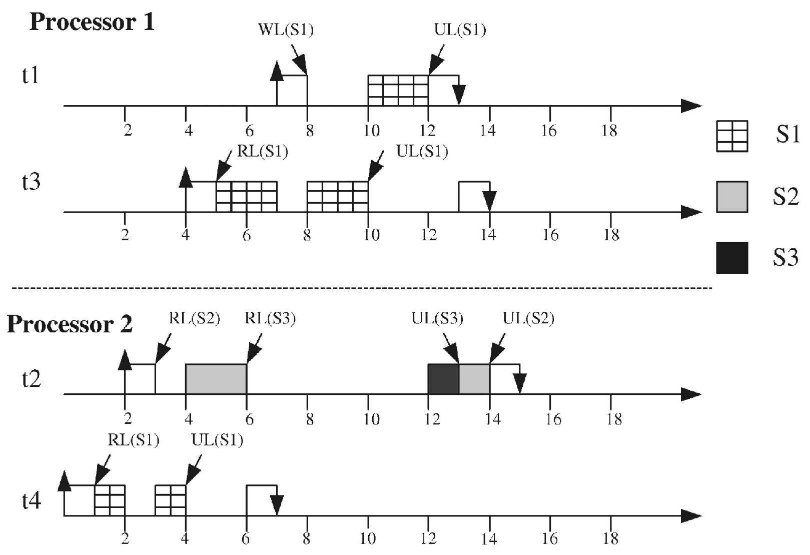

Fig. 1. An RWPCP schedule: $WL(S_i), RL(S_i)$, and $UL(S_i)$ denote the write-lock request, the read-lock request, and the unlock request of data object S_i, respectively.

URTDB or DRTDB may not fit the needs of real-time database systems in a multiprocessor environment because of a potentially low concurrency degree and/or an increased number of priority inversions. We shall show in the following example that the well-known uniprocessor protocol RWPCP may result in a large number of priority inversions in a multiprocessor environment.

Example 1. A RWPCP schedule in a multiprocessor environment.

Let the system consist of four transactions τ_1, τ_2, τ_3, and τ_4 in a two-processor environment, where τ_1 and τ_3 execute on the first processor and τ_2 and τ_4 execute on the second processor. Let the priorities of τ_1, τ_2, τ_3, and τ_4 be 1, 2, 3, and 4, respectively, where 1 is the highest and 4 is the lowest. Suppose that τ_1 may write data object S_1 and τ_3 and τ_4 may read from data object S_1. τ_2 may read from data objects S_2 and S_3. According to the definitions of ceilings, the write priority ceiling WPL_1 and the absolute priority ceiling APL_1 of S_1 are both equal to the priority of τ_1, i.e., 1. The absolute priority ceiling APL_2 of S_2 and the absolute priority ceiling APL_3 of S_3 are both equal to the priority of τ_2, i.e., 2. Because no transaction might write data objects S_2 and S_3, the write priority ceiling WPL_2 of S_2 and the write priority ceiling WPL_3 of S_3 are both equal to 5, which is lower than the priority of all transactions in the system. Note that write priority ceilings are used to resolve read-write or write-write conflict. Since no transaction might write data objects S2 and S3, their write priority ceilings are set as low as possible. (See Fig. 1.)

At time 0, τ_4 arrives at the second processor and starts execution. At time 1, τ_4 read-locks S_1 successfully and $RWPL_1 = WPL_1 = 1$. At time 2, τ_2 arrives and preempts the execution of τ_4 on the second processor. The read-lock request of τ_2 on data object S_2 is blocked at time 3 and τ_4 resumes its execution. At time 4, τ_4 unlocks S_1 and τ_2 is waked up and preempts the execution of τ_4. The read-lock request of S_2 succeeds at time 4 and $RWPL_2 = WPL_2 = 5$. At time 4, τ_3 arrives and starts execution on the first processor. At time 5, the read-lock request of τ_3 on data object S_1 is granted and $RWPL_1 = WPL_1 = 1$ because the priority of τ_3 is higher than $RWPL_2 = 5$. However, the read-lock of τ_3 creates a priority inversion for τ_2 at time 6 when τ_2 issues a read-lock request on data object S_3. It is because the priority of τ_2 is not higher than $RWPL_1 = 1$. The read-lock request of τ_2 on data object S_3 cannot be granted until τ_3 and τ_1 (which arrives at the first processor later at time 7) unlock S_1, although τ_3 may inherit the priority of τ_2 when the blocking happens at time 6.

The above example shows that the number of priority inversions for transactions scheduled by RWPCP may be more than one when there are more than one processor in the system. We must point out that if τ_2 in Example 1 later tries to issue another read-lock on some other data object (after time 12), then τ_2 may experience another priority inversion again if some other lower-priority transaction happens to have read-locked S_1 on the first processor at that time. In other words, the maximum number of priority inversions for transactions scheduled by RWPCP may only be bounded by the number of data objects in the system or become unbounded if a transaction is allowed to lock the same semaphore for many times. It is definitely not acceptable for many time-critical systems. Although the above phenomenon could also happen in distributed real-time databases (DRTDB) theoretically, the locking protocols considered in this paper would not be used in DRTDB because of network delay. In general, distributed real-time databases consist of loosely coupled computers and do not share memory. Researchers usually do not assume the existence of global locks for concurrency control across computers.

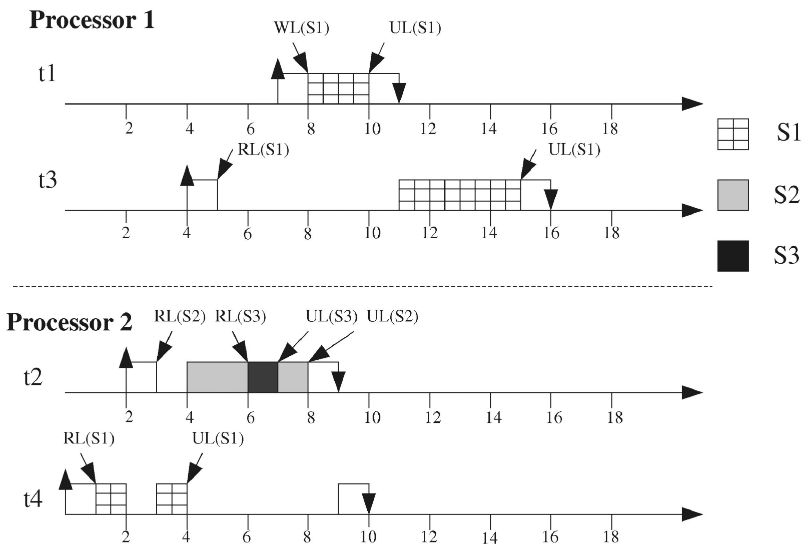

Fig. 2. A 1PI-RWPCP schedule.

4.2 Concurrency Control Protocol with Priority Caps

4.2.1 One-Priority-Inversion RWPCP

The purpose of this section is to propose the concept of priority cap to transform RWPCP into a protocol which guarantees that no transaction will be blocked by more than one transaction in a multiprocessor environment. The concept of priority cap is motivated by the following observation:

Suppose that a higher-priority transaction τ_H locks a data object O_i under RWPCP in a multiprocessor environment. Let τ_H execute on a processor P_1 and another lower-priority τ_L which arrives later on another processor P_2 issue a lock request on another data object O_j. There are two cases to consider:

1. O_i is read-locked by τ_H.

   Because O_i is read-locked by τ_H, the read/write priority ceiling $RWPL_i$ of O_i is set as WPL_i according to the definitions of RWPCP, where WPL_i is the highest priority of transactions which may write O_i. If the priorities of τ_H and τ_L are both higher than WPL_i, then τ_L may successfully lock O_j. If the read/write priority ceiling $RWPL_j$ of O_j is set as a value higher than the priority of τ_H, then τ_H may be blocked by τ_L later when τ_H tries to lock any other data object. On the other hand, if the priority of τ_L is not higher than WPL_i, then τ_H may never be blocked by τ_L because τ_L may not be able to lock O_j unless O_i is unlocked by τ_H.
2. O_i is write-locked by τ_H.

   Because O_i is write-locked by τ_H, the read/write priority ceiling $RWPL_i$ of O_i is set as APL_i which is the highest priority of transactions which may read or write O_i. Since the priority of τ_H is not higher than APL_i, APL_i is higher than the priority of τ_L. In other words, τ_H may never be blocked by τ_L because τ_L will not be able to lock O_j successfully unless O_i is unlocked by τ_H.

The blocking of τ_H by τ_L is caused by the priority gap between the priority of τ_H and the write priority ceiling WPL_i of the data objects locked by τ_H. In a uniprocessor environment, such blocking will not occur under RWPCP because a lower-priority transaction which arrives later will have no chance to lock a data object to block a higher-priority transaction which arrives earlier. In order to prevent the blocking of τ_H by τ_L, the read/write priority ceiling $RWPL_i$ of O_i should be set as a value no less than the priority of τ_H. We call the priority of τ_H as the *locker priority cap* of data object O_i. Note that the locker priority cap of a data object is set dynamically and is equal to the priority of the transaction which locks the data object.

The setting of the read/write priority ceiling $RWPL_i$ is modified as follows: When data object O_i is read-locked by a transaction τ, the read/write priority ceiling $RWPL_i$ of O_i is equal to the maximum of the current $RWPL_i$, WPL_i, and the priority of τ, where the priority of τ is the locker priority cap of data object O_i. (If a data object O_j is not locked by any transaction, $RWPL_j$ is *null* and considered to be lower than the lowest priority level in the system.) When data object O_i is write-locked, the read/write priority ceiling $RWPL_i$ of O_i is equal to APL_i. RWPCP with the modified principles of read/write priority ceiling setting is called *One-Priority-Inversion RWPCP* (1PI-RWPCP). Each transaction should lock data objects in a two-phase locking (2PL) scheme, as required by RWPCP, to enforce the serializability of transaction executions. We shall illustrate 1PI-RWPCP by the following example:

Example 2. A 1PI-RWPCP schedule in a multiprocessor environment.

Let the system consist of four transactions τ_1, τ_2, τ_3, and τ_4 in a two-processor environment, as shown in Example 1. Fig. 2 shows the executions of τ_1, τ_2, τ_3, and τ_4 from time 0 to time 17.

At time 0, τ_4 arrives at the second processor and starts execution. At time 1, τ_4 read-locks S_1 successfully and $RWPL_1 = \max(WPL_1, 4) = 1$. (Note that the priority of τ_4 is 4.) At time 2, τ_2 arrives and preempts the execution of τ_4 on the second processor. The read-lock request of τ_2 on data object S_2 is blocked at time 3 and τ_4 resumes its execution. At time 4, τ_4 unlocks S_1 and τ_2 is waked up and preempts the execution of τ_4. The read-lock request of S_2 succeeds at time 4 and $RWPL_2$ is set as the maximum of WPL_2 and the priority of τ_2, which is equal to 2. At time 4, τ_3 arrives and starts execution on the first processor. At time 5, the read-lock request of τ_3 on data object S_1 is blocked because the priority of τ_3 is no higher than $RWPL_2 = 2$. Note that the locker priority cap of S_2 prevents τ_3 from obtaining any other data object to block τ_2. τ_2 successfully read-locks S_3 at time 6 and commits at time 9. The read-lock request of τ_3 on data object S_1 is granted until time 11.

Example 2 shows that the maximum number of priority inversions under 1PI-RWPCP is reduced to one because a lower-priority transaction which arrives later and executes on another processor now cannot lock a data object to block a higher-priority transaction. The concept of priority cap introduces a new kind of blocking, called *cap blocking*, in multiprocessor environments. For example, the blocking of the read-lock request of τ_3 at time 5 in Example 2 makes the first processor idle for two time units. It is the price paid for the management of priority inversions. We must emphasize that the concept of priority cap can be applied to many other protocols in a similar way in the multiprocessor environments.

4.2.2 Properties

We shall show other important properties of the 1PI-RWPCP protocol as follows:

Definition 1 [20]. *A transitive blocking is said to occur if a (higher-priority) transaction is directly blocked by another (lower priority) transaction which, in turn, is directly blocked by the other (further lower-priority) transaction.*

Lemma 1. *No transitive blocking is possible for transactions scheduled by 1PI-RWPCP in a multiprocessor environment.*

Proof. Suppose that a transaction τ_L directly blocks another transaction τ_M and τ_M directly blocks the other transaction τ_H, where τ_H and τ_L have the highest and lowest priority levels among the three transactions, respectively. Suppose that τ_L locked $z_{L,j}$ which blocks τ_M before τ_M locks $z_{M,k}$ which blocks τ_H. Because $z_{L,j}$ blocks τ_M, it is not possible for τ_M to lock $z_{M,k}$ to block τ_H. In other words, τ_M must lock $z_{M,k}$ which blocks τ_H before τ_L lock $z_{L,j}$ which blocks τ_M. Since $z_{M,k}$ can block τ_H, it can also block τ_L from locking $z_{L,j}$. Therefore, the assumption in transitive blocking is not possible. That is, it is not possible that τ_L directly blocks τ_M and τ_M directly blocks τ_H. □

Theorem 1. *1PI-RWPCP schedules are deadlock-free in a multiprocessor environment.*

Proof. Since Lemma 1 shows that 1PI-RWPCP schedules do not have transitive blocking, a deadlock can only consist of two transactions if it exists. Suppose that there is a deadlock of two transactions τ_H and τ_L such that τ_L directly blocks τ_H, and τ_H directly blocks τ_L, where the priority of τ_H is higher than that of τ_L. Suppose that τ_L lock $z_{L,k}$ which blocks τ_H before τ_H lock $z_{H,j}$ which blocks τ_L. Since $z_{L,k}$ blocks τ_H, it should prevent τ_H from lock $z_{H,j}$ to block τ_L. Suppose that τ_H lock $z_{H,j}$ which blocks τ_L before τ_L lock $z_{L,k}$ which blocks τ_H. Since $z_{H,j}$ blocks τ_L, it should prevent τ_L from locking $z_{L,k}$ to block τ_H. In other words, no deadlock is possible under 1PI-RWPCP. □

Note that the deadlock-freeness of RWPCP can also be proven in a similar way as that in Lemma 1 and Theorem 1.

Theorem 2 [21]. *RWPCP schedules are serializable in a uniprocessor environment.*

Theorem 3. *RWPCP schedules are serializable in a multiprocessor environment.*

Proof. Since a multiprocessor environment will not change the compatibility relationship of read and write operations and the two-phase-locking scheme of transactions scheduled by RWPCP, the correctness of this theorem follows directly from Theorem 2. □

Theorem 4. *1PI-RWPCP schedules are serializable in a multiprocessor environment.*

Proof. Since the set of 1PI-RWPCP schedules is a subset of the set of RWPCP schedules, the correctness of this theorem follows directly from Theorem 3. □

Theorem 5. *The maximum number of priority inversions for any real-time transaction under 1PI-RWPCP in a multiprocessor environment is one.*

Proof. Let a transaction τ_H be directly blocked by two lower-priority transactions τ_M and τ_L. Note that Lemma 1 shows that there is no transitive blocking, let the priority of τ_M be higher than the priority of τ_L. It is obvious that τ_L and τ_M must enter a critical section to block τ_H before τ_H enters any critical section or locks $z_{H,i}$; otherwise, the read/write priority ceiling $RWPL_i$ of the corresponding data object of $z_{H,i}$ will be higher than the priorities of τ_L and τ_M because $RWPL_i$ is higher than the priority of τ_H. Since the priority of τ_M is higher than the priority of τ_L, τ_L must lock $z_{L,j}$ which blocks τ_H before τ_M lock $z_{M,k}$ which blocks τ_H; otherwise, the read/write priority ceiling $RWPL_k$ of the corresponding data object of $z_{M,k}$ will be higher than the priority of τ_L. Since $z_{L,j}$ blocks τ_H, it must also block τ_M. In other words, τ_M cannot lock $z_{M,k}$ to block τ_H and the maximum number of priority inversions for any real-time transaction τ_H under 1PI-RWPCP in a multiprocessor environment is one. □

Let $\Pi = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of periodic transactions (with period p_i and computation requirement c_i) listed in the increasing order of their priorities and HPC_i be the set of transactions with a priority no lower than τ_i in Π and executing on the same processor as τ_i. Suppose that the priority assignment of transactions follows the rate monotonic priority assignment scheme [17] that assigns transactions priorities inversely proportional to their periods. Note that, since the maximum number of priority inversions for any real-time transaction under 1PI-RWPCP in a multiprocessor environment is one, the worst case blocking time b_i of transaction τ_i is the longest duration of any critical section which may block τ_i.

Theorem 6 [17]. *A transaction τ_i scheduled by the rate monotonic scheduling algorithm (RMS) [17] will always meet its deadline for all process phases if*

$$\sum_{j \in HPC_i} \left(\frac{c_j}{p_j}\right) \leq m(2^{\frac{1}{m}} - 1),$$

where m is the number of transactions in HPC_i.

The rate monotonic scheduling (RMS) algorithm [17] is an optimal fixed-priority scheduling algorithm for any independent transaction set which does not share data among transactions.

Theorem 7. *A transaction τ_i scheduled by 1PI-RWPCP will always meet its deadline for all process phases if*

$$\sum_{j \in HPC_i} \left(\frac{c_j}{p_j}\right) + \frac{b_i}{p_i} \leq m(2^{\frac{1}{m}} - 1),$$

where b_i is the worst case blocking time of transaction τ_i and m is the number of transactions in HPC_i.

Proof. By considering the blocking time b_i as the extra computation time for τ_i, the correctness of this theorem directly follows Theorem 6. □

4.3 Two-Version Database and Priority Caps

4.3.1 One-Priority-Inversion Two-Version RWPCP

The purpose of this section is to explore the concurrency level of transaction executions by means of data versions. We shall extend a uniprocessor variant of RWPCP called *Two-Version Priority Ceiling Protocol* (2VPCP) [11] to multiprocessor environments: The rationale behind the multiprocessor 2VPCP called *One-Priority-Inversion 2VPCP* (1PI-2VPCP) is to have more transactions executing concurrently on different processors. Note that concurrently executing transactions may access the same data object in some conflicting mode because there are two versions for each data object.

Let each data object in the system have two versions: consistent version and working version, where a consistent version contains a data value updated by a committed transaction and a working version contains a data value updated by an uncommitted transaction. 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 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 lock compatibility matrix of 1PI-2VPCP is shown in Table 1.

TABLE 1
The Lock Compatibility Matrix of 1PI-2VPCP

request \ locked	Read-Lock	Write-Lock	Certify-Lock
Read-Lock	yes	yes	no
Write-Lock	yes	no	no
Certify-Lock	no	no	no

The transaction which has the highest priority among all ready transactions in a processor is assigned the processor. In order to bound the maximum number of priority inversions in the system, we adopt the idea of priority cap: When a transaction $\tau_{i,j}$ 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 maximum (locker) priority cap of O_k and the highest read/write priority ceilings of the data objects which are locked by other transactions. There are three cases to consider:

1. If $\tau_{i,j}$ requests a read lock on O_k, then the read/write priority ceiling $RWPL_k$ is set as the maximum of the current $RWPL_k$, WPL_k, and the priority of $\tau_{i,j}$
2. If $\tau_{i,j}$ requests a write lock on O_k, then the read/write priority ceiling $RWPL_k$ of data object O_k is set as the maximum of the current $RWPL_k$ and WPL_k. Note that the priority of $\tau_{i,j}$ is no more than WPL_k.
3. If $\tau_{i,j}$ requests a certify lock on O_k, then the read/write priority ceiling $RWPL_k$ 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 higher than the priority of $\tau_{i,j}$.

If the priority of $\tau_{i,j}$ is no higher than the minimum priority cap of O_k and the highest read/write priority ceilings of the data objects which are locked by other transactions, then the lock request is blocked. We shall illustrate 1PI-2VPCP with an example:

Example 3. A 1PI-2VPCP schedule:

Let the system consist of five transactions τ_1, τ_2, τ_3, τ_4, and τ_5 in a two-processor environment, where τ_1, τ_3, and τ_5 execute on the first processor and τ_2 and τ_4 execute on the second processor. Let the priorities of τ_1, τ_2, τ_3, τ_4, and τ_5 be 1, 2, 3, 4, and 5, respectively, where 1 is the highest and 5 is the lowest. Suppose that τ_1 may write data object S_1 and τ_3 and τ_4 may read from data object S_1. τ_2 may read from data object S_2. τ_5 may write data object S_3 and τ_2 and τ_4 may read from data object S_3. According to the definitions of ceilings, the write priority ceiling WPL_1 and the absolute priority ceiling APL_1 of S_1 are both equal to the priority of τ_1, i.e., 1. The absolute priority ceiling APL_2 and the write priority ceiling WPL_2 of S_2 are equal to 2 and 6, where 6 is lower than the priorities of all transactions in the system. The write priority ceiling WPL_3 and the absolute priority ceiling APL_3 of S_3 are equal to 5 and 2, respectively. (See Fig. 3.)

At time 0, τ_5 arrives at the first processor. τ_5 then write-locks S_3 at time 1 and

$$RWPL_3 = \max(WPL_3,\ priority\ (\tau_5)) = 5.$$

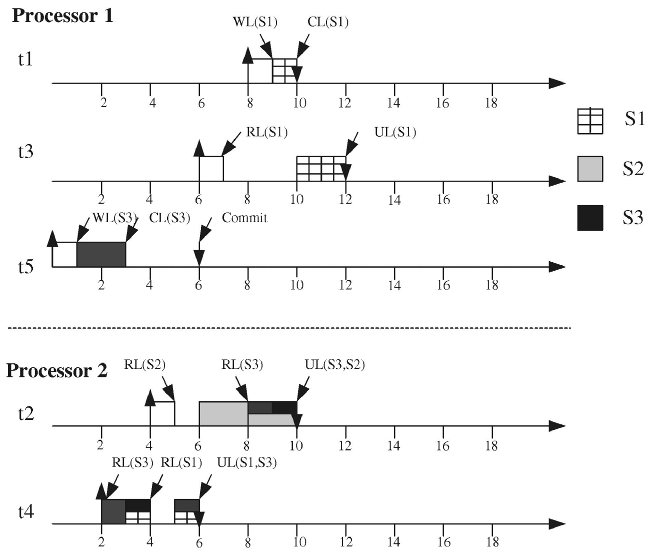

Fig. 3. A 1PI-2VPCP schedule: $WL(S_i), RL(S_i)$, $CL(S_i)$, and $UL(S_i)$ denote the write-lock request, the read-lock request, the certify-lock request, and the unlock request of data object S_i, respectively.

At time 2, τ_4 arrives at the second processor and read-locks S_3 successfully because the priority of τ_4 is higher than $RWPL_3$. Note that data object S_3 is now both read-locked and write-locked by transactions executing on different processors! $RWPL_3$ is set as

$$\max(current\ RWPL_3,\ priority(\tau_4)) = 4.$$

At time 3, τ_4 read-locks S_1 successfully for the same reason and $RWPL_1 = \max(WPL_1, priority(\tau_4)) = 1$. However, the certify-lock request of τ_5 on data object S_3 at time 3 is blocked because the priority of τ_5 is no higher than $RWPL_1$, where S_1 is read-locked by τ_4. At time 4, τ_2 arrives at the second processor and preempts τ_4 on the second processor. At time 5, the read-lock request of τ_2 on data object S_2 is blocked because the priority of τ_2 is no higher than $RWPL_1 = 1$. When S_1 and S_3 are both unlocked at time 6 by τ_4, τ_5 resumes and certify-locks S_3. Suppose that it takes no time for τ_5 to certify-lock S_3 and commits where τ_5 unlocks S_3 when it commits. At time 6, τ_2 read-locks S_2 and

$$RWPL_2 = \max(WPL_2,\ priority(\tau_2)) = 2.$$

τ_3 arrives at the first processor at time 6. The read-lock request of τ_3 on data object S_1 at time 7 is blocked because $RWPL_2$ is set as the maximum of $WPL_2 = 6$ and the priority of τ_2, which is equal to 2. Such blocking prevents the introducing of another priority inversion for τ_2 at time 8. However, we must point out that such blocking also forces the first processor to go in to idle for one time unit. It is the price paid for the priority inversion management.

Note that 1PI-2VPCP improves the performance of 1PI-RWPCP by means of two data versions for each data object. The lock compatibility matrix of 1PI-2VPCP, as shown in Table 1, allows a higher-priority transaction to preempt a lower-priority transaction, even when any read/write conflict occurs. Such preemption reduces the possibility of priority inversion and could improve the schedulability of critical transactions. The consistency of the database is enforced by only permitting transactions to read from the consistent versions of data under the two-phase locking scheme. Theorem 10 provides a more detailed discussion on why 1PI-2VPCP schedules are serializable in a multiprocessor environment.

4.3.2 Properties

We shall show other important properties of the 1PI-2VPCP protocol as follows:

Lemma 2. *No transitive blocking is possible for transactions scheduled by 1PI-2VPCP in a multiprocessor environment.*

Proof. This lemma can be proven in the same way as the proof of Lemma 1. □

Theorem 8. *1PI-2VPCP schedules are deadlock-free in a multiprocessor environment.*

Proof. This theorem can be proven in an analogous way as the proof of Theorem 1, given the fact that no transitive blocking is possible (Lemma 2). □

Theorem 8 [11]. *2VPCP schedules are serializable in a uniprocessor environment.*

Theorem 10. *1PI-2VPCP schedules are serializable in a multiprocessor environment.*

Proof. Since the set of 1PI-2VPCP schedules is a subset of the set of 2VPCP schedules and a multiprocessor environment will not change the compatibility relationship of read and write operations and the two-phase-locking scheme of transactions scheduled by 2VPCP, the correctness of this theorem follows directly from Theorem 9. □

Theorem 11. *The maximum number of priority inversions for any transaction scheduled by 1PI-2VPCP is one.*

Proof. This theorem can be proven in the same way as the proof of Theorem 6. □

Let $\Pi = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of periodic transactions (with period p_i and computation requirement c_i) listed in increasing order of their priorities and HPC_i be the set of transactions with a priority no lower than τ_i in Π and executing on the same processor as τ_i. Suppose that the priority assignment of transactions follows the rate monotonic priority assignment scheme [17]. Note that, since the maximum number of priority inversions for any real-time transaction under 1PI-2VPCP in a multiprocessor environment is one, the worst case blocking time b_i of transaction τ_i is the longest duration of any critical section which may block τ_i.

Theorem 12. *A transaction τ_i scheduled by 1PI-2VPCP will always meet its deadline for all process phases if*

$$\sum_{j \in HPC_i} \left(\frac{c_j}{p_j}\right) + \frac{b_i}{p_i} \leq m\left(2^{\frac{1}{m}} - 1\right),$$

where b_i is the worst case blocking time of transaction τ_i and m is the number of transactions in HPC_i.

Proof. By considering the blocking time b_i as the extra computation time for τ_i, the correctness of this theorem directly follows Theorem 6. □

5 Implementation Issues

The implementation of the concepts for the priority cap and the two-version data requires new data structures and additional runtime overheads. Before their implementation is discussed, we shall first illustrate the implementation issues of the Priority Ceiling Protocol (PCP) and the Read/Write PCP (RWPCP).

PCP could be implemented simply by a stack and a priority queue [20], where each semaphore is assigned a priority ceiling, as defined under PCP. Whenever a semaphore is locked, its corresponding priority ceiling is pushed onto the top of the stack. When a semaphore is unlocked, its corresponding priority ceiling is popped out of the stack. The priority ceiling at the top of the stack is called the *system priority ceiling*. If the priority of a transaction that requests a lock is not higher than the system priority ceiling, then the transaction is blocked and pending on the priority queue. The blocked transaction is said to be directly blocked by the transaction that locks the semaphore with the system priority ceiling. The priority inheritance mechanism would be activated if the priority of the blocked transaction is higher than the priority of any transaction that blocks the blocked transaction. Because there is no transitive blocking, the overhead of priority inheritance is limited.

RWPCP is an extension of PCP: The implementation of RWPCP is similar to that of PCP except that the semaphore corresponding to a data object is now given a write priority ceiling and an absolute priority ceiling [21]. When a semaphore (or the corresponding data object) is locked, its read/write priority ceiling is set according to the definitions of RWPCP. A stack and a priority queue are needed and operate in the same way as that for PCP except that the read/write priority ceiling of the locked semaphore is pushed onto and popped out of the stack. The priority inheritance mechanism operates in the same way as that for PCP and the overheads are the same for PCP and RWPCP.

The implementation of the concept of the priority cap only changes the assignment rule of the read/write priority ceiling of the locked semaphore. That is, when a semaphore (or the corresponding data object) is read-locked by a transaction, the read/write priority ceiling of the locked semaphore is set as the maximum of the priority of the transaction and the write priority ceiling of the semaphore. Because the rest of the RWPCP with the concept of the priority cap (i.e., 1PI-RWPCP) is the same as that for RWPCP, the additional runtime overhead in the implementation of the concept of the priority cap is very limited. The implementation of the concept of the two-version data again changes the assignment rule of the read/write priority ceiling of the locked semaphore and introduces additional overheads in copying data from the updated working version to the consistent version. The major overheads of the RWPCP with the two-version data (i.e., 2VPCP) comes from the data copying because the rest of 2VPCP is the same as that for RWPCP (except the assignment rule of the read/write priority ceiling of the locked semaphore). The copying overhead is proportional to the number of writes in the system. Besides, more space is needed to store the consistent version and the working version of every data object. That is the price paid for the high concurrency of the system.

6 Performance Evaluation

6.1 Performance Metrics

The experiments described in this section are meant to assess the ideas of priority cap and two-version data access in transaction processing for multiprocessor real-time database systems. The simulation experiments compare the performance of the Read/Write Priority Ceiling Protocol (RWPCP) [21], Two-Version Priority Ceiling Protocol (2VPCP) [11], 1PI-RWPCP, and 1PI-2VPCP in multiple processor environments. Note that, although the idea of priority cap can guarantee one priority inversion in the system, it also introduces a new kind of blocking, called *cap blocking*, in multiprocessor environments. The experiments shall evaluate the potential performance degradation in transaction processing due to priority cap. We shall also assess the capability of the proposed methodology in improving the performance of a multiprocessor real-time database system.

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 $\frac{miss_i}{num_i}$. Another metric is the average number of priority inversions experienced by a process, referred to as $PINumber$. Let num_i and pin_i be the total number of process requests (excluding process instances whose absolute deadlines are over the simulation time) and the number of priority inversions experienced by a transaction during an experiment, respectively. $PINumber$ is calculated as $\frac{pin_i}{num_i}$.

In the simulation experiments, a transaction set is a set of transactions which are executing on different processors. Each transaction is a template of its instances and each instance will be instantiated for every request of the transaction. A transaction may issue a sequence of database operations (reads or writes). The purpose of the experiments is to evaluate the capability of the proposed methodology in a multiprocessor real-time database system under different workloads, database sizes, and processor configurations. We shall assess the potential performance degradation in transaction processing due to the implementation of the priority cap concept and the performance improvement because of the idea of the two-version data. The simulation parameters were chosen so that the capability of the proposed concepts would be observed over systems with different numbers of processors, various workloads, and different degrees of access conflict.

The test data sets were generated by a random number generator. The number of processor was two or four. The number of transactions per processor was randomly chosen between 10 and 15. The number of transactions per transaction set was in the range $(20, 60)$, where a transaction

TABLE 2
Parameters of Simulation Experiments, where ROT Stands for Read-Only Transactions

parameter	value
processor number	2 or 4
transaction number per processor	$(10, 15)$
CPU utilization factors	$(60\%, 95\%)$
transaction period	$(10, 10000)$
the number of data objects read by a ROT	$(1, 5)$
the number of data objects read or written by an update transaction	$(1, 5)$
simulation time	$1,000,000$

set is a set of transactions. The experiments started with transaction sets of a utilization factor equal to 60 percent and increased by 5 percent at a time until the utilization factor was equal to 95 percent, 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. Each transaction set was simulated from time 0 to time $1,000,000$. Over 100 transaction sets per utilization factor were tested and their results were averaged. The period of a transaction was randomly chosen in the range $(10,\ 10,000)$. The deadline of a transaction was equal to its period. We did not choose to have preperiod or postperiod deadlines for transactions because such setting did not help in evaluating the capability of the proposed concepts. The priority assignment of transactions follows the rate monotonic priority assignment [17]. The utilization factor of each transaction was no larger than 30 percent 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 5. The critical sections of a transaction in locking data objects were evenly distributed and properly nested for the duration of the transaction. Each transaction only unlocked their write-locks at the commit time to avoid dirty reads in the system. Note that this paper aims at critical real-time systems, such as the avionics example [12] or satellite control systems [3], which have no more 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 2.

6.2 Experimental Results

Figs. 4a and 4b show the miss ratios of the entire transaction set and the top 1/4 priority transactions in the set, respectively, when there were two processors and the database consisted of 50 data objects. It was shown that 1PI-2VPCP and 2VPCP performed better than the other protocols in multiprocessor environments. The reason why 1PI-2VPCP and 2VPCP performed better than the others was because two-version data had a much higher concurrency degree in the multiprocessor environment, especially when workload was heavy loaded. In other words, there was more flexibility in handling conflicting read and write activities of transactions. There was also better concurrency in transaction executions.

It was also interesting to see that the miss ratios of the entire transaction set and even the high priority (top 1/4 priority) transactions under 1PI-2VPCP(/1PI-RWPCP) were better than that under 2VPCP(/RWPCP), as shown in Fig. 4. It was because the protocols with the one priority inversion guarantee mechanism could reduce the maximum number of priority inversions in a multiprocessor environment and, thus, improve the chance of transactions in meeting their deadlines. The average number of priority inversions experienced by transactions under RWPCP (and 2VPCP) was higher than that under 1PI-RWPCP (and 1PI-2VPCP), as shown in Fig. 5. We do not include the figures for the average number of priority inversions when the database size was equal to 100, 200, and 400 and the number of processors was equal to four because they were similar to that in Fig. 5. In the simulation experiments, the maximum number of priority inversions experienced by transactions under RWPCP and 2VPCP could be as large as 7, when

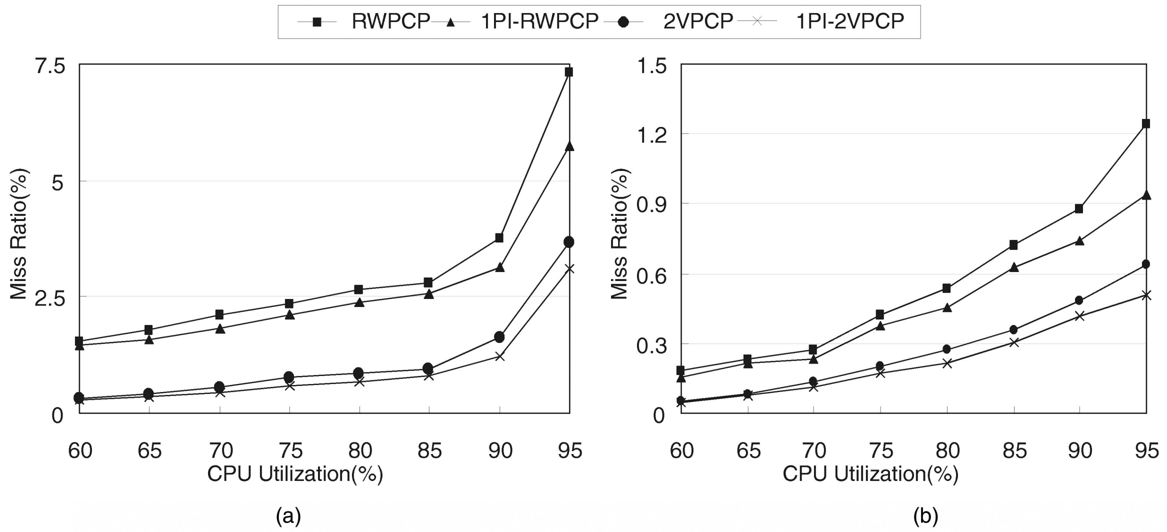

Fig. 4. The miss ratios of RWPCP, 1PI-RWPCP, 2VPCP, and 1PI-2VPCP when there were two processors and the database size was equal to 50. (a) The entire transaction. (b) The top 1/4 priority transactions.

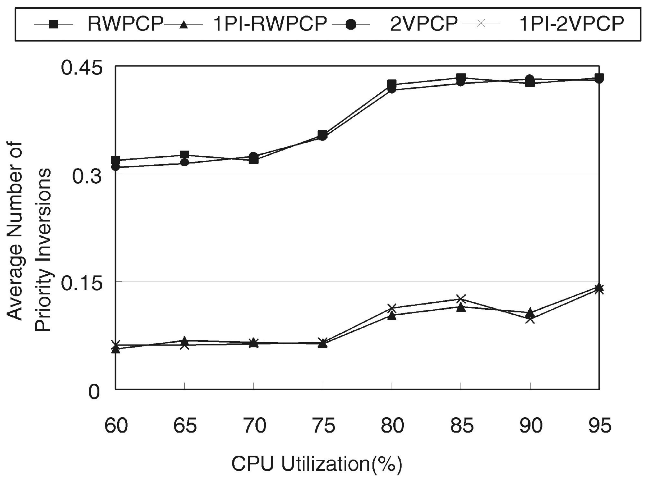

Fig. 5. The number of priority inversions of RWPCP, 1PI-RWPCP, 2VPCP, and 1PI-2VPCP when there were two processors and the database size was equal to 50.

there were two processors and the database size was equal to 50. Note that the maximum number of priority inversions experienced by transactions under 1PI-RWPCP and 1PI-2VPCP was only one.

Figs. 6, 7, and 8 show the miss ratios of the entire transaction set and the top 1/4 priority transactions under RWPCP, 1PI-RWPCP, 2VPCP, and 1PI-2VPCP when there were two or four processors and the database consisted of 50 or 400 data objects. The results in Figs. 6, 7, and 8 were similar to Fig. 4. In the experiment results, when database size increased and the number of processors was the same, the miss ratios of transactions under RWPCP, 1PI-PWPCP, 2VPCP, and 1PI-2VPCP decreased. It was because, when the number of data objects increased, the possibility of access conflict decreased, as shown in Figs. 4 and 6 (7 and 8).

On the other hand, when there were more processors in the system, the miss ratios of transactions under RWPCP, 1PI-PWPCP, 2VPCP, and 1PI-2VPCP were higher, compared to those with a smaller number of processors in the system, as shown in Figs. 4 and 7 (6 and 8). It was because the competition in data access was high and there were potentially more transactions that tried to access the same data object at the same time. The results for the database size equal to 100 and 200 were not included because they were similar to those in Figs. 4, 6, 7, and 8. Although Figs. 4, 6, 7, and 8, did show that two data versions significantly improved the miss ratio of transactions under all kinds of workloads, the system had to maintain two versions (i.e., consistent version and working version) for each data object under 2VPCP and 1PI-2VPCP. It was clear that data versions introduced a higher concurrency degree of transaction executions without increasing the maximum number of priority inversions in the system. The average numbers of access conflicts for a transaction under RWPCP, 1PI-RWPCP, 2VPCP, and 1PI-2VPCP were 0.24, 0.26, 0.16, and 0.17, respectively.

7 Conclusion

This paper explores issues in concurrency control of multiprocessor-based real-time database systems. In particular, we are interested in techniques which can bound the number of priority inversions and, at the same time, better utilize abundant computing power in multiprocessor environments. We show that the maximum number of priority inversions for transactions scheduled by uniprocessor protocols can be much more than one in a multiprocessor environment. The serious priority inversion problem is then solved by the proposed concept of priority cap, which can guarantee one priority inversion in a multiprocessor environment. We then explore a high concurrency degree of transaction executions by means of data versions without increasing the maximum number of priority inversions in the system. The performance of the proposed methodology in transaction processing is validated by a series of simulation experiments. The idea of priority cap is even shown to improve the performance of the system when there are a lot of access conflicts in the system. It is because the benefit of priority cap in reducing

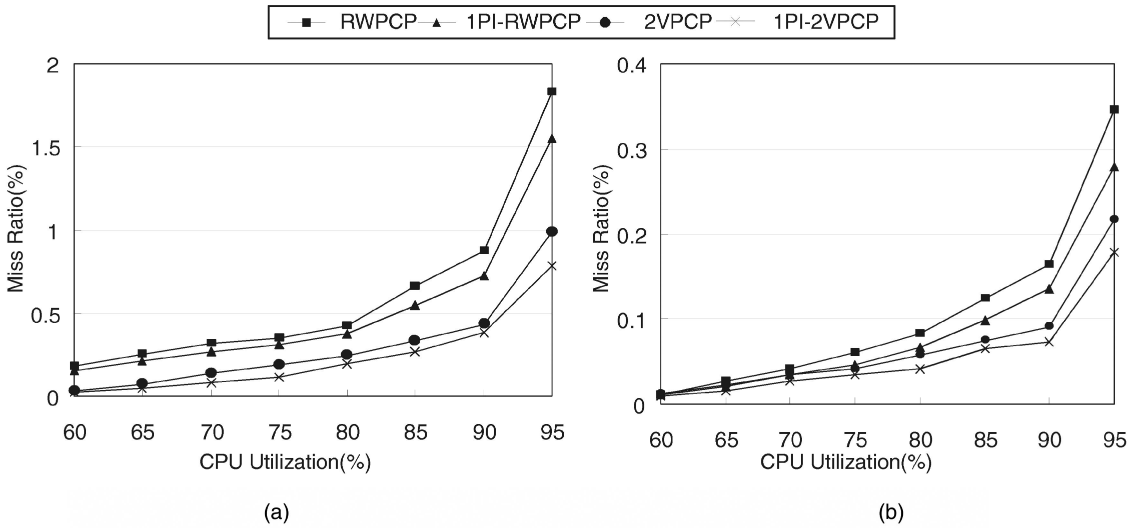

Fig. 6. The miss ratios of RWPCP, 1PI-RWPCP, 2VPCP, and 1PI-2VPCP, when there were two processors, and the database size was equal to 400. (a) The entire transaction set. (b) The top 1/4 priority transactions.

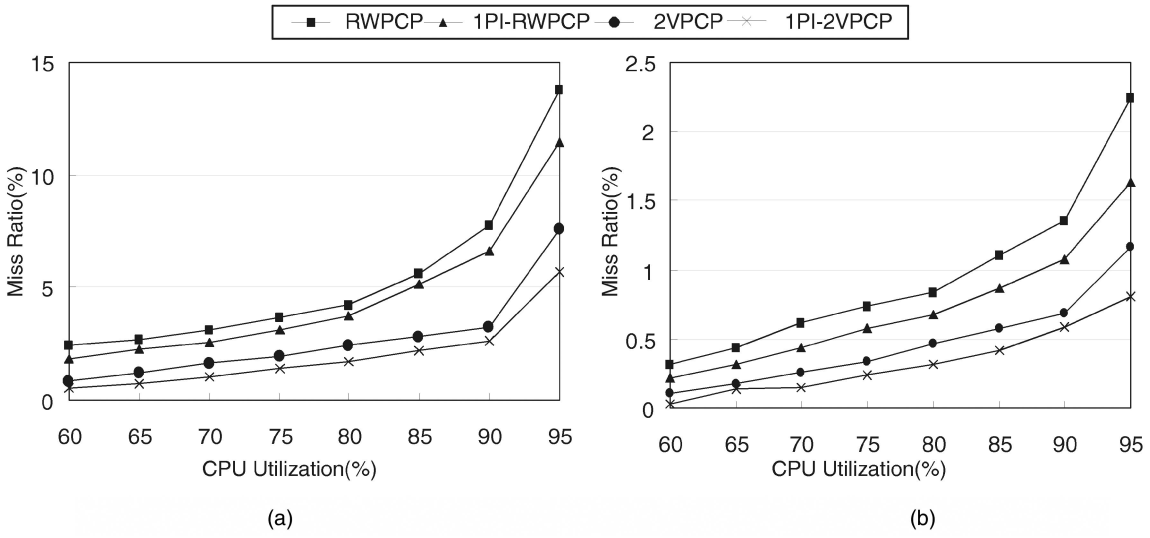

Fig. 7. The miss ratios of RWPCP, 1PI-RWPCP, 2VPCP, and 1PI-2VPCP when there were four processors and the database size was equal to 50. (a) The entire transaction set. (b) The top 1/4 priority transactions.

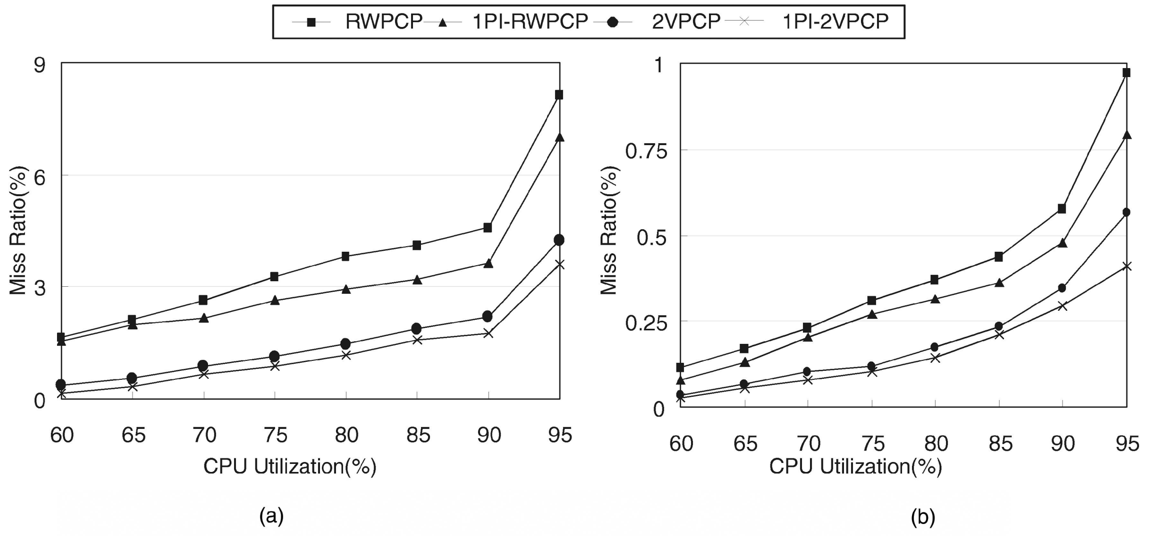

Fig. 8. The miss ratios of RWPCP, 1PI-RWPCP, 2VPCP, and 1PI-2VPCP when there were four processors and the database size was equal to 400. (a) The entire transaction set. (b) The top 1/4 priority transactions.

the blocking time of urgent transactions is far over the loss of committed less urgent transactions, due to the introducing of cap blocking. The idea of two-version data also greatly improves the system performance because of a much higher concurrency degree in the multiprocessor environment. We must emphasize that the techniques proposed in this paper can be applied to many other protocols. RWPCP is merely used to present the ideas proposed in this paper.

Although many high-performance computer systems are now multiprocessor-based, little work has been done in real-time concurrency control of multiprocessor-based real-time database systems. For future research, we shall further explore issues in transaction scheduling in multiprocessor real-time database systems. We should also consider transaction migration and reallocation in the future work to provide more flexibility in transaction scheduling and explore application semantics to further increase the concurrency degree of the system.

Acknowledgments

This research was supported in part by a grant from the Chung-Shan Institute of Science and Technolgy and a research grant from the National Science Council under grant NSC89-2213-E-194-011. This paper is an extended version of a paper in the 12th Euromicro Conference on Real-Time Systems, Sweden, 2000. The authors further extend the results in exploring a high concurrency degree of transaction executions by means of data versions, without increasing the maximum number of priority inversions in the system. More complete simulation experiments and proofs have been included.

References

[1] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
[2] A. Bestavros, "Timeliness via Speculation for Real-Time Databases," *Proc. IEEE 15th Real-Time Systems Symp.,* 1994.

[3] A. Burns, A.J. Wellings, C.M. Bailey, and E. Fyfe, "A Case Study in Hard Real-Time System Design and Implementation," Technical Report YCS 190, Dept. of Computer Science, Univ. of York, 1993.

[4] M.P. Bodlaender, S.A.E. Sassen, P.D.V. van der Stok, and J. van der Wal, "The Response Time Distrubution in a Multi-Processor Database with Single Queue Static Locking," *Proc. Fourth Int'l Workshop Parallel and Distributed Real-Time Systems*, pp. 118-121, 1996.

[5] L. Chih, "Optimistic Similarity-Based Concurrency Control," *Proc. IEEE 1998 Real-Time Technology and Applications Symp.*, June 1998.

[6] A. Chiu, B. Kao, and K.-y. Lam, "Comparing Two-Phase Locking and Optimistic Concurrency Control Protocols in Multiprocessor Real-Time Database," *Proc. Int'l Workshop Parallel and Distributed Real-Time Systems*, pp. 141-148, 1997.

[7] L.B.C. Dipippo and V.F. Wolfe, "Object-Based Semantic Real-Time Concurrency Control," *Proc. IEEE Real-Time Systems Symp.*, Dec. 1993.

[8] J.R. Haritsa, M.J. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proc. Ninth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pp. 331-343, Apr. 1990.

[9] J.R. Haritsa, M.J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proc. IEEE 11th Real-Time Systems Symp.*, 1990.

[10] T.-W. Kuo and A.K. Mok, "SSP: A Semantics-Based Protocol for Real-Time Data Access," *Proc. IEEE 14th Real-Time Systems Symp.*, Dec. 1993.

[11] T.-W. Kuo, Y.-T. Kao, and L.C. Shu, "A Two-Version Approach for Real-Time Concurrency Control and Recovery," *Proc. Third IEEE Int'l High Assurance Systems Eng. Symp.*, Nov. 1998.

[12] C.D. Locke, D.R. Vogel, and T.J. Mesler, "Building a Predictable Avionics Platform in Ada: A Case Study" *Proc. IEEE 12th Real-Time Systems Symp.*, Dec. 1991.

[13] K-W. Lam and S.L. Hung, "A Preemptive Transaction Scheduling Protocol for Controlling Priority Inversion," *Proc. Third Int'l Workshop Real-Time Computing Systems and Applications*, Oct. 1996.

[14] 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," *Proc. IEEE 19th Real-Time Systems Symp.*, Dec. 1998.

[15] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proc. IEEE 11th Real-Time Systems Symp.*, Dec. 1990.

[16] V.B. Lortz, K.G. Shin, and J. Kim, "MDARTS: A Multiprocessor Database Architecture for Hard Real-Time Systems," *IEEE Trans. Knowledge and Data Eng.*, vol. 12, no. 4, July/Aug. 2000.

[17] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[18] E. Navathe, *Fundamentals of Database Systems, second ed.* Addison-Wesley, 1994.

[19] C.-S. Peng and K.-J. Lin, "A Semantic-Based Concurrency Control Protocol for Real-Time Transactions," *Proc. IEEE 1996 Real-Time Technology and Applications Symp.*, 1996.

[20] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Technical Report CMU-CS-87-181, Dept. of Computer Science, Carnegie Mellon Univ., Nov. 1987 , and *IEEE Trans. Computers*, vol. 39, no. 9, Sept. 1990.

[21] L. Sha, R. Rajkumar, S.H. Son, and C.H. Chang, "A Real-Time Locking Protocol," *IEEE Trans. Computers*, vol. 40, no. 7, July 1991.

[22] L. Sha, R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, vol. 17, 1988.

[23] S.H. Son and C.H. Chang, "Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment," *Proc. Int'l Conf. Distributed Computing Systems*, 1990.

[24] M. Xiong, K. Ramamritham, R. Sivasankaran, J.A. Stankovic, and D. Towsley, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *Proc. IEEE Real-Time Systems Symp.*, pp. 240-251, Dec. 1996.

[25] P. Xuan, S. Sen, O. Gonzalez, J. Fernandez, and K. Ramamritham, "Broadcast on Demand: Efficient and Timely Dissemination of Data in Mobile Environments," *Proc. IEEE Real-Time Technology and Applications Symp.*, 1997.

Tei-Wei Kuo received the BSE degree in computer science and information engineering from National Taiwan University in Taipei, Taiwan, in 1986. He received the MS and PhD degrees in computer sciences from the University of Texas at Austin in 1990 and 1994, respectively. He is currently a professor in the Department of Computer Science and Information Engineering of the National Taiwan University, Taiwan, Republic of China. He was an associate professor in the Department of Computer Science and Information Engineering of the National Chung Cheng University, Taiwan, from August 1994 to July 2000. His research interests include real-time databases, real-time process scheduling, real-time operating systems, and control systems. He was the program cochair of the IEEE Seventh Real-Time Technology and Applications Symposium, 2001 and has served as an associate editor of the *Journal of Real-Time Systems* since 1998. He has consulted for government and industry on problems in various real-time systems design. Dr. Kuo is a member of the IEEE and the IEEE Computer Society.

Jun Wu received the BSE degree in computer science and information engineering from I-Shou University in Kaohsiung, Taiwan, in 1996. He received the MS degree from the Management Information Department of the National Yunlin University of Science and Technology in 1998. He is currently a PhD student in the Department of Computer Science and Information Engineering of the National Chung Cheng University in Chiayi, Taiwan, Republic of China. His research interests include real-time process scheduling, real-time databases, and location management of moving objects.

Hsin-Chia Hsih received the Bachelor degree in computer science and information engineering from the National Chung Cheng University, Chiayi, Republic of China, in June 1997. He received the Master degree in computer science and information engineering from the same university in June 1999. His research interest is in real-time concurrency control and real-time process scheduling. He is now in research and development.