The Impacts of Write-Through Procedures and Checkpointing on Real-Time Concurrency Control

TEI-WEI KUO[1], YEN-HSI HOU[2] AND KAM-YIU LAM[3]

[1]*Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 106, Republic of China*
[2]*Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan 621, Republic of China*
[3]*Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong*
Email: ktw@csie.ntu.edu.tw

In this paper, we study the impacts of checkpointing and write-through procedures, which are critical in maintaining database recoverability and transaction durability, on the performance of a well-known real-time concurrency control protocol, the Read/Write Priority Ceiling Protocol (RWPCP). Although RWPCP can guarantee the schedulability of real-time transactions, it could be unrecoverable, and the priority inversion problems could be unbounded, when transaction commitment is considered. In this paper, we first propose to extend RWPCP with deferred-commitment and extended-locking-period methods to resolve the problems. Then, we study the impacts of different checkpointing granularities and checkpointing methods on the proposed recoverable RWPCP. A detailed simulation study was conducted to evaluate the performance of the recoverable RWPCP with different checkpointing methods under various workloads.

Received 8 February 2002; revised 27 June 2002

1. INTRODUCTION

Real-time concurrency control has been an active research topic over the last decade. Researchers have proposed efficient concurrency control protocols for various types of real-time database systems to maintain database consistency, and at the same time to meet response-time requirements of real-time transactions [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. In particular, Higher-Priority Two Phase Locking (HP-2PL) [11] was proposed for soft and firm real-time database systems to minimize the blocking time of higher-priority transactions by restarting lower-priority transactions. The Read/Write Priority Ceiling Protocol (RWPCP) [9] was proposed for hard real-time database systems, which often have well-defined data and execution requirements, to guarantee the deadlines of hard real-time transactions.

Recoverability and transaction durability have been important research topics in non-real-time main-memory database systems over the past few decades [12, 13, 14]. A *write-through* procedure must be done to enforce the updating of transaction logs into stable storage [12, 15]. Researchers have proposed various efficient logging techniques and mechanisms for transaction and system recovery [16, 17, 18], such as fuzzy checkpointing [19], action-consistent (AC) and transaction-consistent (TC) checkpoints [20]. In order to eliminate the impact of unpredictable I/O delays on system performance, real-time transactions are often assumed to be executed in a main-memory database system [13]. The issues in maintaining transaction durability, such as those under checkpointing and write-through procedures, are often ignored in the design of real-time concurrency control protocols and in their performance evaluation [13, 19, 20]. We must point out that, although many data items in a real-time database system might become invalid as the time passes by, the durability issue can still not be ignored because the unavailability or the corruption of certain data items (even for a short period of time) may result in various fatal system failures [21, 22, 23]. It is important to explore the interactivities of real-time concurrency control and the mechanisms in maintaining transaction durability, especially when the mechanisms could significantly increase the lock holding time and, thus, the priority inversion time of critical transactions.

In this paper, we explore the impacts of the mechanisms for the durability guarantee on real-time concurrency control, in particular those under checkpointing and write-through procedures. We choose the RWPCP in the study because it is well-known to the community of real-time database research, and also guarantees a bounded number of priority inversions and is deadlock-free. How to guarantee transaction deadlines is critical to the design of hard real-time database systems since any deadline missing a hard real-time transaction could be fatal to the entire system. We first show that schedules generated

from RWPCP could be irrecoverable when transaction commitment is considered. We then revise RWPCP by two simple techniques, i.e. *deferred commitment* and *extended locking period* [12, 15], and provide discussions on their impacts on priority inversion and deadlock-freeness. Checkpointing is then modelled as a periodic real-time transaction under the framework of RWPCP. Different methods and granularities of checkpointing under different variants of RWPCP are considered. We show that different checkpointing granularities do impose different impacts on the adopted concurrency control protocol, e.g. the data conflict probability and the priority inversion time. A detailed simulation program is implemented to evaluate the impacts of different checkpointing methods on the performance of recoverable RWPCP. We must point out that the recoverability problems of RWPCP can also be observed in many other real-time locking protocols, such as HP-2PL and many priority-inheritance-related methods [24], etc., although there could be different degrees of impact. The results of this work on deferred commitment, extended locking period and checkpointing could also be applied to many other lock-based protocols.

The rest of the paper is organized as follows. Section 2 summarizes the related work. Section 3 defines the system model and provides research motivation. Section 4 proposes ways to revise RWPCP to make it recoverable and discusses their impacts on priority inversion and deadlock-freeness. In Section 5, we model the checkpointing process as a real-time transaction under RWPCP. Different granularities for checkpointing are considered. Section 6 reports the simulation model and the experimental results which show the impacts of the different checkpointing methods on RWPCP. Section 7 summarizes our conclusions.

2. RELATED WORK

Researchers have proposed various efficient real-time concurrency control protocols to reduce the number of deadline violations and to maintain database consistency, e.g. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Most of the proposed protocols are mainly based on transaction restart and data reservation. For example, HP-2PL [11] resolves the problem of priority inversion by restarting lower-priority transactions when the lock request of a higher-priority transaction is in conflict with a lower-priority transaction. The restart method was also proposed for optimistic concurrency control (OCC) [25]. Various priority-cognitive conflict resolution methods were proposed for use in the OCC method based on the forward validation scheme. Although transaction restart is effective in resolving data access conflict, it is, in general, only good for soft and firm real-time database systems since the deadlines of real-time transactions can hardly be guaranteed. Lower-priority transactions may be repeatedly restarted and consequently miss their deadlines.

Many other real-time concurrency control protocols are based on the idea of data reservation, in which the Priority Ceiling Protocol (PCP) [26] is one of the well-known examples. PCP is a lock-based protocol. In PCP, the idea of priority inheritance is adopted to reduce the blocking time of a higher-priority transaction. When a higher-priority transaction is blocked by a lower-priority transaction, the priority of the latter is raised to that of the former. PCP could guarantee that the maximum number of priority inversions for each transaction is no more than one. The guarantee is very important because it provides performance guarantees to critical transactions in meeting their deadline requirements. RWPCP is an extension of PCP [9]. While PCP only allows exclusive locks on data objects, RWPCP explores read and write semantics of transaction operations in improving the concurrency without sacrificing the schedulability of transactions. Similar to PCP, no deadlock is possible for RWPCP schedules and the maximum number of priority inversions for any transaction is one [9]. Although various highly efficient real-time concurrency control protocols have been proposed, many of them, such as HP-2PL, PCP and RWPCP, often result in irrecoverable schedules if the transactions do not follow the strict 2PL principles in lock handling.

Logging, checkpointing and recovery have been important research topics for non-real-time main-memory database systems over the past few decades [16, 17, 19, 20, 27]. In particular, Hagmann [19] proposed fuzzy checkpointing for recovery in main-memory databases, where fuzzy checkpointing requires only a little synchronization with executing transactions. Fuzzy checkpointing flushes (dirty) pages from the main memory to the secondary storage, where locks and other transaction activities are ignored. Salem and Garcia-Molina [20] compared several checkpointing algorithms with different degrees of consistency of the backup copy they produced. They considered fuzzy, AC and TC checkpointing, where a TC backup reflects transaction activities atomically and each action, e.g. a record update, in an AC backup is reflected atomically. A ping-pong backup scheme which uses two backup disk-resident databases alternatively for checkpointing is used. Woo *et al.* [18] explored logical logging under fuzzy checkpointing, where a logical log entry describes a higher-level operation, e.g. 'a record is inserted into a file F and F's indices are updated'. Logical logging can change a long physical log into a record of only a few words [12]. Li and Eich [17] examined and presented post-crash log processing for fuzzy checkpointing algorithms under major updating paradigms in main-memory databases: immediate updating and deferred updating. Levy and Silberschatz [16] proposed a powerful incremental approach for recovery processing in main-memory databases. An incremental restart algorithm and a checkpoint-like technique called log-driven backups were proposed to operate in an incremental manner, in parallel with transaction processing, where a restart procedure is invoked after a system crash to restore the database to its most recent consistent state. Log-driven backups use log records as the means for propagating updates to the backup database rather than relying on page flushes.

Although a lot of research works have been done on logging and recovery for traditional database systems, little work has explored logging and recovery for real-time

database systems [21, 22]. Two of the previous studies in the area are [21, 22]. In [22], Sivasankaran *et al.* proposed a partitioned logging and recovery algorithm for real-time disk-resident databases. The logs are partitioned based on data classes, such as critical and temporal, to enhance parallel logging and recovery. Non-volatile RAM-based devices are then used to reduce the unpredictability of real-time databases. In [21], Huang and Gruenwald extended the idea of partition checkpointing [28] to real-time main-memory databases. They derived checkpoint frequencies for different database partitions based on the idea that temporal data of short validity intervals should be checkpointed more frequently than persistent data, since the value of a temporal data object changes with time and must be kept up-to-date [21, 22]. Another related work is [29] in which a new real-time commit protocol which allows transactions to 'optimistically' borrow uncommitted prepared data in a controlled manner was proposed to minimize the number of missed transaction deadlines. However, to the best of our knowledge, little work has been done until now on issues related to the performance of real-time concurrency control when durability is considered.

3. SYSTEM MODEL AND RESEARCH MOTIVATION

3.1. System model

We consider real-time memory-resident database systems with stable storage for the durability guarantees. A transaction is a sequence of read and write operations and it may arrive periodically or aperiodically. Each *periodic transaction* is associated with a fixed period and a deadline. Each periodic transaction should be completed no later than its deadline which is defined as its arrival time plus its period. A *sporadic transaction* is not a periodic transaction and it is associated with a deadline and a minimum separation time. The inter-arrival time of any two successive sporadic transactions from the same source should not be less than the *minimum separation time* and each transaction should be completed no later than its deadline.

In general, the behaviour and characteristics of transactions in a real-time database system are more predictable and well-defined than those in a traditional database system [23, 30]. Each transaction in a real-time database system with stringent response-time requirements is usually associated with a worst-case computation time [26]. However, when a soft real-time database system is considered, the computation time of a transaction might be hard to assess. A proper concurrency control protocol is still needed to minimize the number of deadline violations. In this paper, we assume that the data requirements of a transaction can be predicted before its execution since this is one of the requirements of RWPCP [9].

We are interested in lock-based concurrency control protocols. There are basically 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, where a data object, in general, may be a field of some record, a record or even a block. In this paper, we assume that a data object is a record and we will use the terms 'data object' and 'record' interchangeably. The setting and release of a lock follow the RWPCP principles (or its variants which will be introduced later). We assume in this paper that the locks of data objects are properly nested. In other words, if the locking operation on a data object is not later than the locking operation of another data object in a transaction, the corresponding unlocking operation of the former data object is not later than the corresponding unlocking operation of the latter data object. Note that this is one of the assumptions of RWPCP when handling the priority inversion problem. If the assumption does not hold, then the maximum number of priority inversions for a transaction under RWPCP will be more than one. We must point out that the assumption does not prevent transactions from having non-overlapping critical sections.

A transaction reaches its *pre-commit point* when all of its operations have been executed successfully and the effects of all of the transaction operations on the database have been recorded in the transaction logs [15]. Before the actual commitment of a transaction, a *write-through* procedure must be done to enforce the updates of the transaction logs into stable storage [15].

We model the checkpointing process as a periodic real-time transaction. Each checkpointing transaction (called a checkpointer) is assigned a priority and consists of a sequence of operations to flush data items from the main memory to stable storage. They will be executed concurrently with the real-time transactions. Different checkpointing techniques may be adopted to investigate the relationship and impacts on the adopted real-time concurrency control protocol.

3.2. Research motivation—the recoverability in the RWPCP

The purpose of this section is to investigate the well-known RWPCP [9] with regard to recoverability issues as a motivation of this research. 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 lock, respectively. The write priority ceiling WPL_i of data object O_i equals the priority of the highest priority of the transactions which may write O_i. The absolute priority ceiling APL_i of data object O_i equals the priority of the highest priority of the 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 set to WPL_i. When data object O_i is write-locked, the read/write priority ceiling $RWPL_i$ of O_i is set to APL_i.

The transaction which has the highest priority among all executing transactions will be assigned to use the processor. The lock request of a transaction τ_i may be granted if its priority is higher than the highest read/write priority ceiling $RWPL_i$ of all the data objects currently locked by other transactions. Otherwise, the lock request is rejected and the transaction will be blocked until its requested lock is

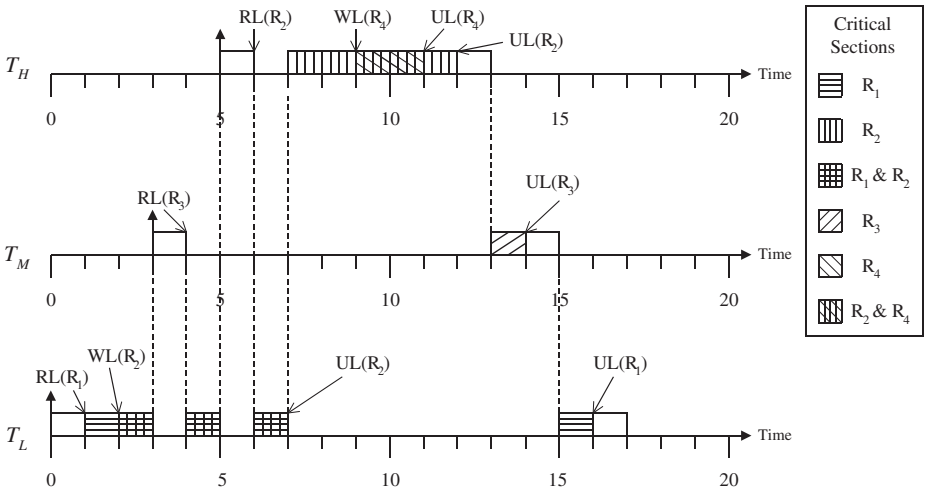

FIGURE 1. A RWPCP schedule.

released by the holder. A transaction τ_i uses its assigned priority for execution and setting locks, unless it is blocking a higher-priority transaction. For this case, a priority inheritance technique is adopted.

EXAMPLE 1. (A RWPCP schedule) We illustrate the principles and the recoverability problem of RWPCP by an example. Suppose that there are three transactions τ_H, τ_M and τ_L in a uniprocessor environment. Let the priorities of τ_H, τ_M and τ_L be 1, 2 and 3, respectively, where 1 is the highest and 3 is the lowest. Suppose that τ_H may read data object R_2 and write data object R_4, and τ_M may read data object R_3. τ_L may read data object R_1 and write data object R_2. According to the definitions of the priority ceilings, the absolute priority ceiling APL_1 of R_1 is 3. The write priority ceiling WPL_2 and the absolute priority ceiling APL_2 of R_2 are 3 and 1, respectively. The absolute priority ceiling APL_3 of R_3 is 2. The write priority ceiling WPL_4 and the absolute priority ceiling APL_4 of R_4 are both 1. Note that the write priority ceilings of R_1 and R_3 are set to a priority level, e.g. 4, which is lower than the priority of any transaction in the system.

Figure 1 describes the execution schedule of τ_H, τ_M and τ_L, where $WL(R_i)$, $RL(R_i)$ and $UL(R_i)$ denote the write-lock, read-lock and unlock requests on data object R_i, respectively. At time 0, τ_L starts execution. At time 1, τ_L read-locks R_1 successfully and $RWPL_1 = WPL_1 = 4$. At time 2, τ_L write-locks R_2 successfully and $RWPL_2 = APL_2 = 1$. At time 3, τ_M arrives and preempts τ_L. At time 4, the read-lock request of τ_M on R_3 is rejected because the priority of τ_M is not higher than $RWPL_2 = APL_2 = 1$. Thus, τ_M is blocked and τ_L resumes its execution at time 4. Note that τ_L inherits the priority of τ_M because τ_L blocks τ_M. At time 5, τ_H arrives and preempts τ_L. At time 6, the read-lock request of τ_H on R_2 is rejected because the priority of τ_H is not higher than $RWPL_2 = APL_2 = 1$. Thus, τ_H is blocked and τ_L resumes its execution at time 6. Note that τ_L inherits the priority of τ_H because τ_L blocks τ_H. At time 7, τ_L unlocks R_2 and τ_L resumes its priority 3. τ_H preempts τ_L and read-locks R_2 successfully at time 7 because the priority of τ_H is higher than $RWPL_1 = 4$. At time 9, τ_H write-locks R_4 and $RWPL_4 = APL_1 = 1$. τ_H unlocks R_4 at time 11. At time 12, τ_H unlocks R_2. At time 13, τ_H commits and then τ_M resumes its execution and read-locks R_3 and $RWPL_3 = WPL_3 = 4$. τ_M unlocks R_3 at time 14 and commits at time 15. Then, τ_L resumes its execution at time 15 and unlocks R_1 at time 16. τ_L commits at time 17.

Note that the RWPCP schedule shown in Figure 1 is not a recoverable schedule. A schedule is recoverable if no transaction τ commits before any transaction from which τ reads commits [15]. In the example, τ_H, which reads from τ_L, commits before τ_L commits. If τ_L aborts after the commitment of τ_H, the schedule is not recoverable [12, 15].

4. RECOVERABLE RWPCP

In this section, we shall study how RWPCP can be revised to make all RWPCP schedules recoverable. In particular, we consider two well-known techniques: deferred commitment and extended locking period [12, 15]. We shall also discuss the impacts of the enhancements on the performance of RWPCP.

4.1. Deferred commitment

A simple way to make a RWPCP schedule recoverable is to defer the commitment of a transaction τ until all transactions from which τ reads commit. We call this variant of RWPCP *RWPCP with Deferred Commitment (RWPCP-DC)*. In order to reduce the delay period, the priority inheritance policy may be adopted for those transactions from which the delayed transaction τ reads. We shall illustrate the proposed mechanism by the following example.

EXAMPLE 2. (RWPCP-DC) Suppose the system consists of the transaction set used in Example 1. For simplicity, suppose that each transaction takes one unit of time to complete the write-through and pre-commit procedures, and the computation time of each transaction in this example is the same as that in Example 1.

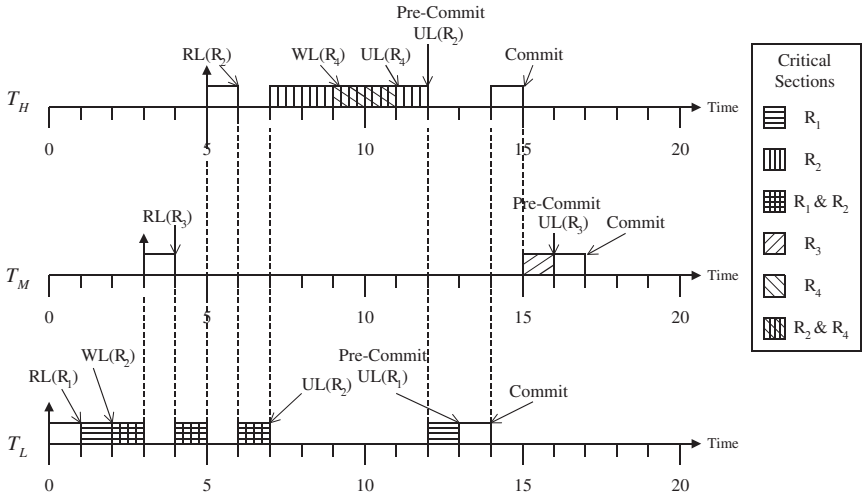

FIGURE 2. A RWPCP schedule with deferred commitment.

Figure 2 shows a schedule when RWPCP-DC is adopted. The schedules in Figures 1 (in Example 1) and 2 are exactly the same from time 0 to time 12. At time 12, when τ_H unlocks R_2 and pre-commits, the commitment of τ_H is deferred. The deferment of commitment causes τ_L to inherit the priority of τ_H and to resume its execution at time 12. τ_L unlocks R_1 and commits at time 14. After τ_L commits, τ_H starts to execute the write-through procedure at time 14 and commits at time 15. τ_M resumes its execution at time 15 and commits at time 17.

The deferred commitment mechanism generates recoverable schedules by reordering the commitment times of transactions. However, the problem of RWPCP-DC is that the maximum number of priority inversions for a transaction becomes unbounded because there may exist a sequence of uncommitted transactions, where each transaction reads from another transaction in the sequence. This can seriously affect the schedulability of the protocol and make the system performance unpredictable.

THEOREM 1. *Transactions scheduled by RWPCP-DC may suffer from an unbounded number of priority inversions.*

Proof. The deferred commitment mechanism generates recoverable schedules by reordering the commitment times of transactions. A transaction cannot commit until all of the transactions which it reads from have committed. Since there may exist a sequence of uncommitted transactions, where each transaction reads from another transaction in the sequence, the length of the commit chain is unlimited. In other words, the maximum number of priority inversions for a transaction under RWPCP-DC is unbounded (see Example 2). □

4.2. Extended locking period

Another way to make RWPCP recoverable is to extend the locking period of the write-locks of a transaction to prevent other transactions from reading dirty data until the commitment of the transaction. We call this variant of RWPCP *RWPCP with Extended Locking Period (RWPCP-ELP)*. Since RWPCP requires nested locking of data objects, RWPCP-ELP should keep the locking of data objects properly nested. We shall illustrate RWPCP-ELP by the following example.

EXAMPLE 3. (RWPCP-ELP) Suppose the system consists of the transaction set used in Examples 1 and 2. We also assume that each transaction takes one unit of time to complete the write-through and pre-commit procedures, and the computation time of each transaction in this example is the same as that in Examples 1 and 2.

Figure 3 shows a schedule when RWPCP-ELP is adopted. From time 0 to time 7, the schedule is the same as that in Figures 1 and 2. At time 7, τ_L delays the release of its write-lock on R_2 until its commitment at time 9. Then, τ_H resumes its execution and read-locks R_2 successfully. τ_H write-locks at time 11 and commits at time 15. At time 15, τ_M resumes its execution. τ_M locks and unlocks R_3 at times 15 and 16, respectively. At time 17, τ_M commits.

Example 3 shows that RWPCP-ELP may significantly reduce the concurrency in transaction execution since the lock holding time is longer. Note that the original RWPCP is a conservative locking protocol. It resolves the problem of data synchronization and minimizes the problem of priority inversion by blocking transactions (with different priority ceilings). Delaying the lock release time under RWPCP-ELP can worsen the problem of transaction blocking and further reduce the degree of concurrency. Unlike RWPCP-DC, the number of priority inversions in RWPCP-ELP is bounded and remains one in the worst case (although the duration of priority inversion time is lengthened). Since the maximum number of priority inversions for RWPCP-ELP is the same as that for the original RWPCP, except that the locking duration is lengthened, it is obvious that RWPCP-ELP remains deadlock-free and all schedules from RWPCP-ELP are serializable.

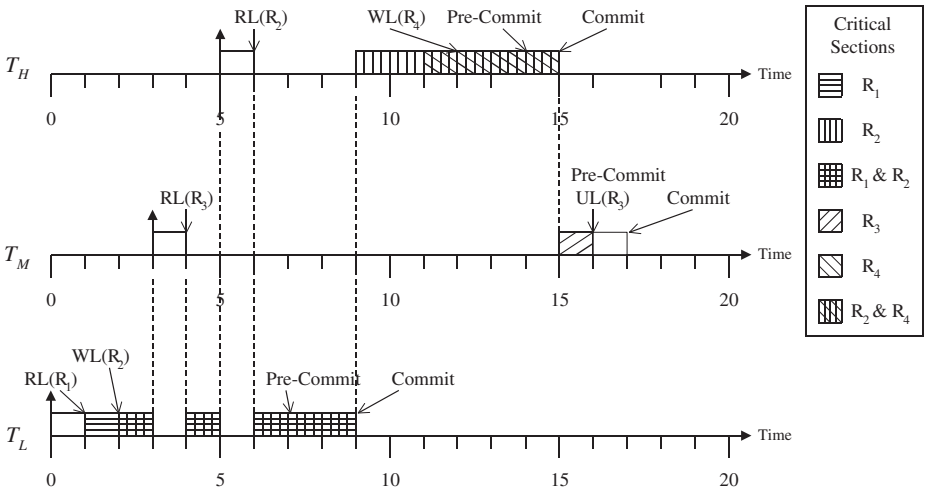

FIGURE 3. A RWPCP schedule with extended locking period.

5. REAL-TIME CHECKPOINTING

While Section 4 provides a study on the recoverability of real-time concurrency control based on RWPCP and its impacts on the schedulability guarantee, the purpose of this section is to address the impacts of different checkpointing methods on RWPCP and our proposed extensions, RWPCP-DC and RWPCP-ELP. We model checkpointing under a real-time scheduling framework using RWPCP. Different checkpointing methods are considered: fuzzy checkpointing [19], record-based checkpointing [20], page-based checkpointing and transaction-consistent checkpointing [20]. The four checkpointing methods represent four different granularities in checkpointing, which are in a word, a data object (record), a page or a segment, where a segment is a collection of pages such that no transaction accesses data belonging to different segments. We do not intend to evaluate the performance of the corresponding recovery mechanisms of the checkpointing methods discussed in this paper. We refer interested readers to [20] for the excellent performance evaluation of different recovery mechanisms.

We must point out that although Sections 4 and 5 cover different aspects of the durability issues on real-time concurrency control, they could be integrated under the scheduling framework of RWPCP, that is, RWPCP-DC and RWPCP-ELP with different checkpointing processes/methods. Note that the real-time checkpointing process is a real-time process under RWPCP. The study on deferred commitment and extended locking period in Section 4 can be directly applied to RWPCP with different checkpointing processes in Section 5.

5.1. Fuzzy checkpointing

Fuzzy checkpointing, which requires only little synchronization with executing transactions, was first suggested for recovery in main-memory databases in [19]. We assume the use of the ping-pong backup scheme, as suggested in [20], where two backup disk-resident databases are used alternatively for checkpointing.

A fuzzy checkpointing process proceeds as follows [19, 20]: when a new fuzzy checkpoint begins, the checkpointer writes a *begin-checkpoint* marker to stable storage. The checkpointer then flushes (dirty) pages from the main memory to stable storage, where locks and other transaction activities are ignored. In other words, transactions may execute and update the database during the flushing of memory contents. The only impact on the processing of real-time transactions is that when the checkpointer is flushing a word, which is also required by some transaction, the transaction needs to wait for the completion of the flushing. The updates may be reflected in the backup and the logs in stable storage. When the checkpointer finishes the backup work, an *end-checkpoint* marker will be written to stable storage. After a system failure, the recovery manager will first read the backup database into the main memory and then apply the logs to the database to restore the database back to a consistent state before the failure.

The first technical question here is what should be the priority level assigned to the checkpointer. An important consideration is that the fuzzy checkpointing method should not affect or should only impose very limited impacts on the execution of any transaction which may access the page which the checkpointer is currently flushing to the secondary storage. A transaction may preempt the checkpointer at any time and access any data objects on the page under fuzzy checkpointing if its priority is higher [19, 20]. At the same time, when the checkpointer is flushing a page which may be accessed by a higher-priority transaction, the checkpointer should be assigned a higher priority in order to finish the flushing earlier.

Let the write priority floor WPF_i of a data object O_i be the lowest priority of the transaction which may write O_i and the write priority floor WPF_k^g of a page P_k be the minimum write priority floor of all data objects residing (entirely or

partially) in the page. Note that the definition of the write priority floor of a page remains even if a data object is stored across some page boundary.

The priority of the checkpointer is dynamically assigned based on the base priority, *BP*, of the checkpointer and the write priority floor WPF_k^g of the page P_k, which the checkpointer is currently flushing to the stable storage such that when the checkpointer is flushing page P_k, its priority is set as the maximum of *BP* and $(WPF_k^g - 1)$. When the checkpointer finishes flushing page P_k, its priority is reset as *BP*.

Using WPF_k^g can minimize the impact on the scheduling of real-time transactions and, at the same time, provide some protection from transactions which might update the page currently being flushed by the checkpointer. If a larger value is assigned to the base priority, *BP*, the system can finish the checkpointing process faster. The value of *BP* can be determined based on the priorities of the real-time transactions in the system. Two simple methods of assigning the value for *BP* are to use the mean and maximum priorities of the real-time transactions. Note that if a higher value is assigned to *BP*, the impact of the checkpointing method on the execution of real-time transactions will be higher. On the other hand, if the value of *BP* is too small, i.e. smaller than all the real-time transactions, the checkpointer can only execute during the idle time of the CPU. This will seriously affect the purposes of the checkpointing process. In the assignment of the value for *BP*, it is important to strike a balance between the interference of the checkpointing method on execution of real-time transactions and at the same time to achieve the purposes of checkpointing. The flushing of a page to the secondary storage can be done either indirectly through the I/O buffer or directly to the secondary storage. If the flushing of a page to stable storage is done indirectly through the I/O buffer, then the checkpointer copies the page to a buffer and initiates the corresponding I/O activities.

In performing the backup process, the checkpointer processes pages in decreasing order of their write priority ceilings. The rationale behind this order is to backup pages which may contain hot or critical data first, in case there is not enough time to backup the entire database. It is assumed that the page which may be accessed by a higher-priority transaction will be more critical. Note that in many kinds of real-time database applications, such as an avionics system [31], there is no need to backup the entire database. We shall show that the maximum number of priority inversions for a real-time transaction due to fuzzy checkpointing is one.

THEOREM 2. *The maximum number of priority inversions, due to fuzzy checkpointing, for a real-time transaction with a priority higher than BP is one.*

Proof. The checkpointer can only block a transaction τ with a priority higher than *BP* when it is currently flushing a page with a higher write priority floor. As soon as the checkpointer finishes the flushing, the checkpointer will be preempted by τ because its priority is reset as *BP*. τ will not be blocked again by the checkpointer because the checkpointer which has a priority lower than τ does not have a chance to start flushing another page before τ terminates. □

Note that real-time transactions which have a priority lower than *BP* will be preempted and blocked by the checkpointer.

THEOREM 3. *RWPCP schedules with fuzzy checkpointing are deadlock-free.*

Proof. Since the checkpointer never locks any record or page and is never blocked by any real-time transaction, the checkpointer will never be involved in a deadlock with any other transactions. Since transactions scheduled by RWPCP are deadlock-free [9, 26], RWPCP schedules with fuzzy checkpointing are deadlock-free. □

5.2. Record-based checkpointing

The idea of record-based checkpointing is similar to the idea of action-consistent checkpointing in [20]. The checkpointer locks and copies one record at a time to the I/O buffer. Before the checkpointer locks the next record, the checkpointer must unlock the currently locked record. Once a buffer is filled up, the direct memory access (DMA) is initiated to copy the buffer contents to the proper location in the secondary storage if DMA is available. Otherwise, disk I/O operations are initiated. Note that DMA shares the workload of the CPU in handling I/O operations. The operating system can set up a high-level I/O operation through DMA and let DMA initiate a number of I/O operations and memory copying. For simplicity, the checkpointer backs up records in the same memory page consecutively and processes one page at a time. The backup order of records in the same page is assumed to be arbitrary. After a system failure, the recovery manager will first read the backup database into the main memory and then apply the logs from stable storage to the database to bring it back to a consistent state [20].

Two technical questions for record-based checkpointing are: (1) the priority assignment to the checkpointer; and (2) the backup order of pages. Similar to fuzzy checkpointing, we propose to dynamically assign priority to the checkpointer. However, unlike fuzzy checkpointing, we use the write priority ceiling of the record which the checkpointer has locked, to determine the priority of the checkpointer instead of the write priority floor. The reason behind this is to try to finish the backup of a record faster since it may be requested by a higher-priority transaction. Note that if the checkpointer has locked a record, no real-time transactions are allowed to access the record.

Let *BP* be the base priority of the checkpointer. When the checkpointer successfully locks a record, its priority is set as the maximum of *BP* and the write priority ceiling of the record. When the checkpointer unlocks a data object, its priority is reset to its original priority, i.e. *BP*. The granting and rejection of a lock request of the checkpointer follow the definitions of RWPCP. However, the priority of the

checkpointer is not considered in the derivation of the read/write priority ceilings of any data object. If the lock request of the checkpointer is blocked by a transaction, then the transaction inherits the priority of the checkpointer.

Let the write priority ceiling WPL_k^g of a page P_k be defined as the maximum write priority ceiling of all data objects residing (entirely or partially) in the page. Similar to fuzzy checkpointing, the checkpointer backs up pages in decreasing order of their write priority ceiling. The purpose is to backup pages which may contain hot and critical data first. We can show that the maximum number of priority inversions for a real-time transaction due to record-based checkpointing is one.

THEOREM 4. *The maximum number of priority inversions, due to record-based checkpointing, for a real-time transaction with a priority higher than BP is one.*

Proof. The checkpointer can only block a transaction τ with a priority higher than *BP* when it is currently locking a record. As soon as the checkpointer releases the lock, the checkpointer will be preempted by τ because its priority is reset back to *BP*. τ will not be blocked again by the checkpointer since the checkpointer which has a priority lower than τ cannot lock any more records after it releases the lock and before τ terminates. □

THEOREM 5. *RWPCP schedules with record-based checkpointing are deadlock-free.*

Proof. Since the checkpointer never holds a record and never requests another record at the same time, the checkpointer will never be involved in a deadlock with any other transactions. Since transactions scheduled by RWPCP are deadlock-free [9, 26], RWPCP schedules with record-based checkpointing are deadlock-free. □

5.3. Page-based checkpointing

The procedures of page-based checkpointing are the same as for record-based checkpointing, except that the checkpointer locks and flushes a whole page, instead of one record, at a time. It is assumed that a page is larger than a record. Locking a page instead of a record can reduce the locking overheads but increase the probability of lock conflict with real-time transactions. After a system failure, the recovery manager will first read the backup database into the main memory and then apply the logs to the database to bring the database back to a consistent state.

Let the write priority ceiling WPL_k^g of a page P_k be defined as the maximum write priority ceiling of all data objects residing (entirely or partially) in the page. The granting and rejection of a lock request from the checkpointer follows the definitions of RWPCP. That is, when the checkpointer wants to lock a page, its priority must be higher than the read/write priority ceilings $RWPL_i$ of all data objects O_i currently locked by transactions in the system. Let *BP* be the base priority of the checkpointer. When the checkpointer successfully locks a page P_k, all data objects in the page are implicitly locked by the checkpointer and its priority is set as the maximum of *BP* and the write priority ceiling WPC_k^g of the page. Before the checkpointer locks another page, the checkpointer has to unlock the currently locked page. When the checkpointer unlocks a page, its priority is reset back to its original priority, i.e. *BP*. If the base priority of the checkpointer is higher, the system can finish the checkpointing process faster. Similar to the record-based checkpointing, the checkpointer backs up pages in decreasing order of their write priority ceilings and the priority of the checkpointer is not considered in the derivation of the read/write priority ceiling of any data object.

THEOREM 6. *The maximum number of priority inversions, due to page-based checkpointing, for a real-time transaction with a priority higher than BP is one.*

Proof. The checkpointer can only block a transaction τ with a priority higher than *BP* when it is currently locking a page. As soon as the checkpointer releases the lock, the checkpointer will be preempted by τ because its priority is reset back to *BP*. τ will not be blocked again by the checkpointer because the checkpointer which has a priority lower than τ cannot lock any more pages after it releases the lock and before τ terminates. □

THEOREM 7. *RWPCP schedules with page-based checkpointing are deadlock-free.*

Proof. Since the checkpointer never holds a page and never requests another page at the same time, the checkpointer will never be involved in a deadlock with any other transactions. Since transactions scheduled by RWPCP are deadlock-free [26, 9], RWPCP schedules with page-based checkpointing are deadlock-free. □

5.4. Transaction-consistent checkpointing

The transaction-consistent checkpointing algorithm discussed in this section is closely related to the idea of transaction-consistent checkpointing in [20]. During a checkpointing process, the checkpointer backs up segments one by one. It locks all pages in a segment in the 2PL fashion and flushes the (dirty) pages from the main memory to the secondary storage, where a segment is a collection of pages such that no transaction accesses data belonging to different segments. If DMA is available, then DMA is initiated for I/O transfers. Otherwise, disk I/O operations are initiated.

Since the checkpointer produces a consistent database image, the recovery procedure after a system failure can be done easily. It is because the updates of a transaction either entirely reflect in the backup or do not reflect at all in the backup [12, 15, 20]. However, the probability of lock conflict with real-time transactions is higher compared with other methods, i.e. fuzzy and record-based checkpointing, since the checkpointer only releases locks in a segment when it has completed the backup of the segment. In order to minimize the impact of the checkpointer on the execution of real-time transactions, the priority of the checkpointer is considered in the derivation of the read/write priority

ceilings of all data objects. The checkpointer can simply be considered as another real-time transaction under the transaction-consistent checkpointing algorithm.

Let *BP* be the base priority of the checkpointer. When the checkpointer wants to lock a page, its priority must be higher than the read/write priority ceilings $RWPL_i$ of all data objects O_i currently locked by other transactions in the system. The priority of the checkpointer will not be raised when it locks a page (as required in record-based or page-based checkpointing), unless the checkpointer inherits the priority from a higher-priority transaction because it is blocking the real-time transaction.

Similar to the three checkpointing methods introduced in the previous subsections, the checkpointer locks pages in a segment in increasing order of their write priority ceilings with the purpose of minimizing the locking period of pages with large write priority ceilings, where the write priority ceiling of a page reflects the maximum priority of the transactions which may update data objects in the page.

THEOREM 8. *The maximum number of priority inversions, due to transaction-consistent checkpointing, for a real-time transaction with a priority higher than BP is one.*

Proof. Since the checkpointer is considered as a transaction under the framework of RWPCP and RWPCP guarantees one priority inversion, the maximum number of priority inversions, due to transaction-consistent checkpointing, for a real-time transaction with a priority higher than *BP* is one. □

THEOREM 9. *RWPCP schedules with transaction-consistent checkpointing are deadlock-free.*

Proof. Since the checkpointer is considered as a transaction under the framework of RWPCP and RWPCP is deadlock-free, RWPCP with transaction-consistent checkpointing is deadlock-free. □

6. PERFORMANCE EVALUATION

6.1. Simulation model, model parameters and performance metrics

The simulation experiments consisted of two parts. We first compared the performance of RWPCP [9], RWPCP-DC and RWPCP-ELP, under different real-time transaction workloads and database sizes. We then investigated the impacts of different checkpointing methods on the system performance when a durable RWPCP, i.e. RWPCP-DC, was used for concurrency control.

The simulation model was developed based on the system model introduced in Section 3. We considered a main-memory-resident real-time database system with flash memory cards as RAM disks for logging. The simulator consisted of a set of transaction generators. One of the generators was responsible for the generation of the checkpointer (for checkpointing and write-through procedures) and the other generators were responsible for generating real-time transactions. A real-time transaction consisted of a sequence of read and write operations on data objects (records). Real-time transactions generated by the same generator had the same data requirements and the same period. We considered firm real-time periodic transaction systems in the experiments. The deadline of a transaction was equal to its arrival time plus its period. When a transaction missed its deadline, it would be aborted. Transactions were assigned priorities according to the rate monotonic priority assignment, where the priorities of transactions were inversely proportional to their periods. Transactions and the checkpointer were scheduled according to the adopted concurrency control protocols.

Table 1 summarizes the model parameters and their baseline values. Note that the purpose of the performance study was not to investigate the performance of the concurrency control protocols for a particular real-time database system. It was to explore the performance characteristics of the proposed protocols, e.g. RWPCP-DC and RCPCP-ELP, and the impacts of different checkpointing methods on their performance under different conditions and settings. The baseline values of the parameters were fundamentally determined based on the previous studies of real-time database systems, e.g. [32], and the purposes of our study. For example, we have done intensive simulations over a wide range of database sizes to evaluate the performance of the protocols under different degrees of data contention. The set of records accessed by a transaction was generated following a normal distribution with a randomized mean and a variance (which was one tenth of the database size). In other words, if the database size was large, the probability of data conflict would be small. We defined the upper and lower period bounds for generation of real-time transactions. The period of a real-time transaction generator was randomly selected in a range between the upper and lower period bounds defined in the table. In the experiments, we varied the lower period bound of real-time transactions to investigate the system performance under different real-time transaction workloads. When the workload was heavy, the system had more difficulty in guaranteeing transaction deadlines. In order to illustrate the performance characteristics and differences of the methods, we tested the system over a wide range of workloads, including a very heavy workload (that might result in a high deadline missing probability of transactions).

In the model, we assumed that a 4-byte unique ID was assigned to each transaction and record. The type of each log entry was stored in one byte, where the type was *begin_transaction*, *commit*, *abort*, *redo* or *undo*. A redo (or undo) log entry consisted of a 4-byte transaction ID, a 4-byte record ID, a 1-byte log entry type and a 128-byte after-image (or before-image) of the corresponding record. A *begin_transaction* (or *commit* or *abort*) log entry consisted of a 4-byte transaction ID and a 1-byte log entry type. Intel Value-Series-100 flash memory cards, which offer high-performance disk emulation, were used to estimate the performance of RAM disks, where 32 bytes per word were assumed, and every word write and read costs 10 μs and 100 ns, respectively [33]. The RAM disks were assumed to

TABLE 1. Parameters and their baseline values (the size of the read set plus the size of the write size is no less than one).

Parameter	Value
The number of real-time transaction generators	10
The upper period bound for generation of real-time transactions	45 ms
The size of the read set	0 to 12 uniformly distributed
The size of the write set	0 to 12 uniformly distributed
CPU computation time for a write operation	0.04 ms
CPU computation time for a read operation	0.0004 ms
The period of the checkpointer	1000 ms
Record size	128 bytes
Page size	640 bytes
The number of records in the database	2048
Transaction ID length	4 bytes
Record ID length	4 bytes
begin_transaction log entry size	5 bytes
Commit or abort log entry size	5 bytes
Undo log entry size	137 bytes
Redo log entry size	137 bytes
Bytes per word of flash memory	32 bytes
Word write time of flash memory	10 μs
Word read time of flash memory	100 ns
Simulation length	20,000 real-time transactions

store the system logs. Since the unit of a write operation was in 32 bytes, each log entry was assumed to be written in multiples of the write units. That is, a redo (or undo) log entry would cost 50 μs to flush into the system logs.

Upon the start of a real-time transaction, a 'begin_transaction' log was written into the log file. Before a transaction updated the database, the corresponding undo and redo log entries needed to be flushed into the system log first. When a transaction committed, the write-through procedure flushed all of the redo log entries of the transaction into the system logs. Then, the system wrote a 'commit transaction' to the log and committed the transaction.

The checkpointer checked up all objects and flushed the new values to the stable storage. When the checkpointer began its execution, it wrote a 'begin checkpointing' log to the log file, and then it processed objects under different strategies, such as fuzzy, record-based, page-based and transaction-consistent methods. Finally, it wrote an 'end checkpointing' log to the log file after the completion of the checkpointing process.

The primary performance metric used was *miss rate*. It was defined as the percentage of deadline violations for real-time transactions over the total number of real-time transactions generated. Miss rate could reflect the probability of deadline missing as well as the degree of guarantee provided by the system. When transactions started missing their deadlines, deadline guarantee could no longer be held. In addition to miss rate, we also measured the probability of lock conflicts experienced by real-time transactions, referred to as *RT-trans conflict rate*, and the probability of lock conflicts experienced by the checkpointer, referred to as *CP to RT-trans conflict rate*. RT-trans conflict rate was defined as the total number of lock conflicts experienced by real-time transactions over the total number of lock requests generated by them. It indicated the probability of lock conflicts and the concurrency in real-time transaction executions. CP to RT-trans conflict rate was defined as the total number of lock conflicts suffered by the checkpointer (due to lock conflicts with real-time transactions) over the total number of lock requests issued by the checkpointer.

Lock conflicts experienced by real-time transactions can be classified into two types: lock conflicts with other real-time transactions and lock conflicts with the checkpointer. A performance metric, referred to as *RT-trans to CP conflict rate*, was defined as the total number of lock conflicts experienced by real-time transactions (due to lock conflicts with the checkpointer) over the total number of locks requested by real-time transactions. The metric indicates the degree of interference for real-time transactions because of the execution of the checkpointer.

6.2. Comparison of RWPCP, RWPCP-DC and RWPCP-ELP

In this set of experiments, we compared the performance of RWPCP-DC and RWPCP-ELP with RWPCP when fuzzy checkpointing was adopted. We investigated their performance under different workloads and database sizes. Although both RWPCP-DC and RWPCP-ELP were recoverable, the probability of lock conflicts under RWPCP-DC and RWPCP-ELP could be higher than that of RWPCP since the lock holding times by transactions might be longer. This was the cost we needed to pay for making the system recoverable. In the figures, we include RWPCP together with RWPCP-DC and RWPCP-ELP for comparison to show the cost to make it recoverable. The reason for

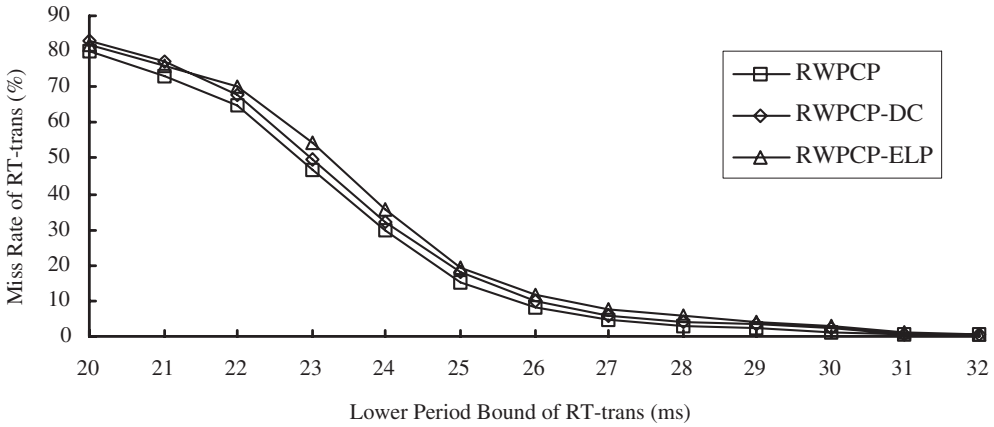

FIGURE 4. The impacts of the lower period bound of real-time transactions on their miss rate under fuzzy checkpointing (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 2048 database objects).

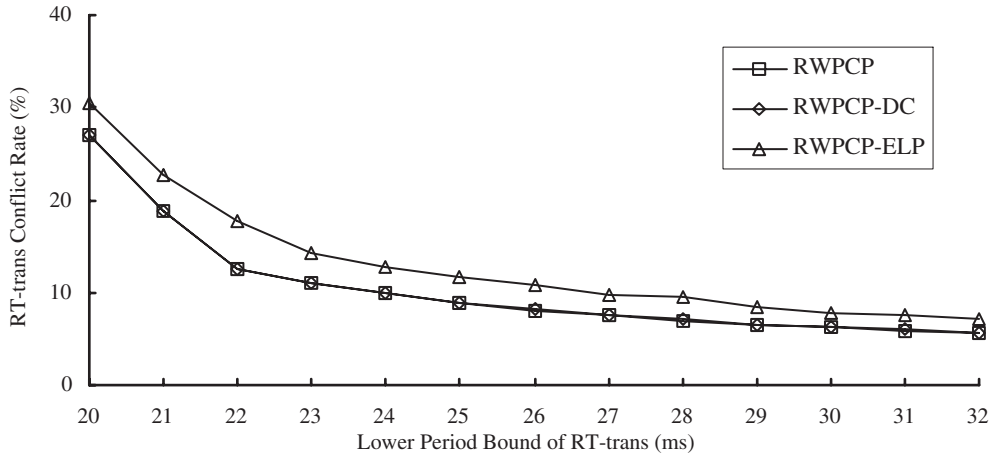

FIGURE 5. The impacts of the lower period bound of real-time transactions on RT-trans conflict rate under fuzzy checkpointing (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 2048 database objects).

choosing fuzzy checkpointing in this part of the experiments was because fuzzy checkpointing had less impact on the performance of real-time concurrency control protocols, compared to other checkpointing methods, so that the performance comparison of different protocols could be more reasonable.

Figure 4 shows the miss rates of transactions under RWPCP, RWPCP-DC and RWPCP-ELP when the workloads of real-time transactions varied with different lower period bounds for real-time transactions. It can be seen from Figure 4 that the miss rate of real-time transactions dropped under all of the three protocols when the value of the lower period bound was increased because of a lower workload. Compared to RWPCP, the miss rate of real-time transactions under RWPCP-ELP was higher for all the tested workloads including heavy workload situations. The higher miss rate of RWPCP-ELP was due to a higher probability of lock conflicts among real-time transactions, as shown in Figure 5. It came from the fact that each transaction scheduled by RWPCP-ELP held its locks until it committed. Such a long duration of lock holding time also resulted in a high system priority ceiling for most of the time and further decreased the concurrency of the system. Many real-time transactions were unnecessarily blocked.

The miss rate of real-time transactions under RWPCP-DC was slightly higher than that under RWPCP when the workload was not heavy and both of the protocols had similar RT-trans conflict rates, as shown in Figure 5. However, the miss rate of real-time transactions under RWPCP-DC became higher than those under RWPCP and RWPCP-ELP when the workload was heavy. This was because of the long blocking time of transactions, due to the unbounded number of priority inversions. When the system workload was heavy, the priority inversion problem seriously affected the performance of RWPCP-DC. When the system workload was low, the probability of long priority inversion time was low.

Figures 6 and 7 show the miss rates and RT-trans conflict rates of real-time transactions under RWPCP, RWPCP-DC and RWPCP-ELP, when the database size was only 512 data objects. The results shown in Figure 6 are similar to those shown in Figure 4 except that the miss rates of real-time transactions (under all of the three protocols) are slightly higher. This is because when the database size is smaller,

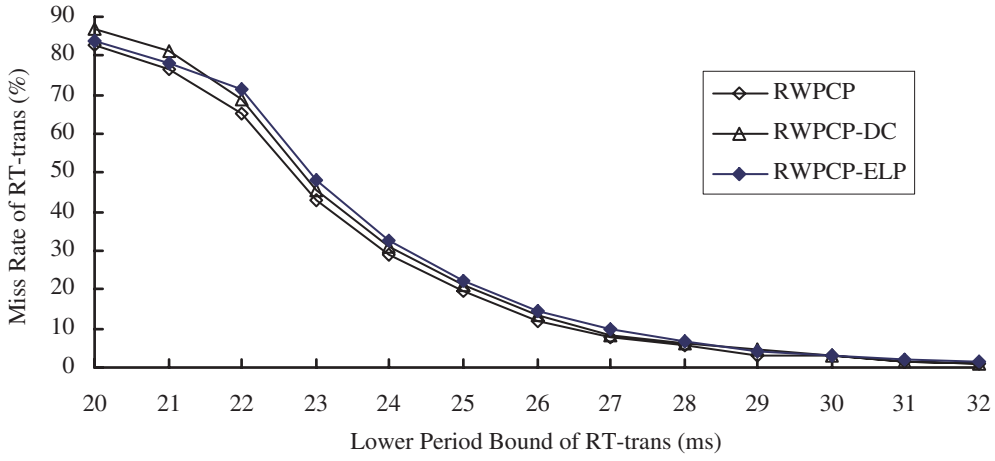

FIGURE 6. The impacts of the lower period bound of real-time transactions on their miss rate under fuzzy checkpointing (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 512 database objects).

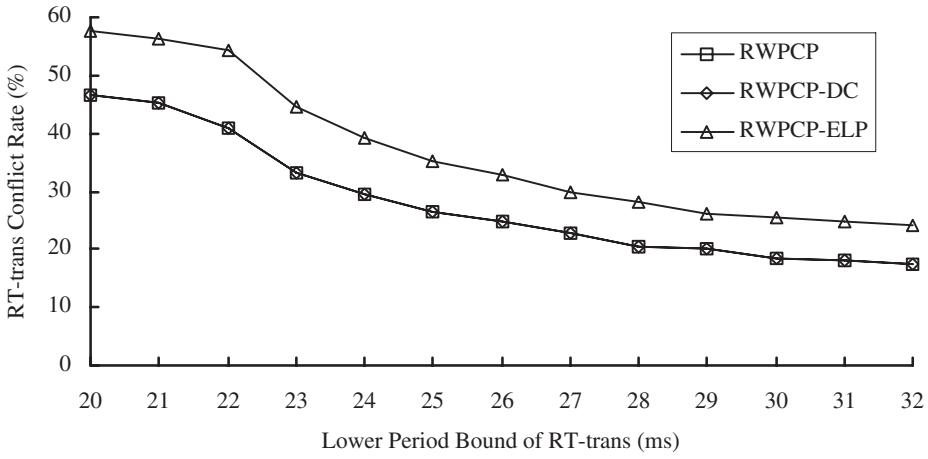

FIGURE 7. The impacts of the lower period bound of real-time transactions on RT-trans conflict rate under fuzzy checkpointing (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 512 database objects).

the lock conflict probability is higher. When the database consisted of 512 data objects, the performance of RWPCP remained the best, compared to RWPCP-DC and RWPCP-ELP. The performance of RWPCP-DC was between RWPCP and RWPCP-ELP, except when the workload was heavy. Similar to the results in Figure 4, the performance of RWPCP-DC was worse than that of RWPCP-ELP under a heavy workload. Additional experiments were also performed for the case when the database consisted of 8192 data objects. The results were consistent with those obtained when the database sizes were 2048 and 512 data objects. The performance differences of all the protocols remained the same.

6.3. Interference of checkpointing and concurrency control

This set of experiments was meant to explore the interference of checkpointing and concurrency control. We shall exploit the performance issues regarding different checkpointing granules and methods under different system workloads and database sizes. In the following figures, we only report the performance of RWPCP-DC in order to make the figures easier to read. The impacts of different checkpointing mechanisms on RWPCP and RWPCP-ELP were similar to the impacts on RWPCP-DC. The reasons behind choosing RWPCP-DC in the presentation of the experimental results were because RWPCP-DC was recoverable and its performance was between those of RWPCP and RWPCP-ELP (as shown in Section 6.2).

Figure 8 shows the miss rates of real-time transactions when different checkpointing methods were used. The base priority *BP* of the checkpointer was set as the average priority of the real-time transactions. As shown in Figure 8, the record-based checkpointing (abbreviated as 'Record' in the figures) gave the smallest miss rate and the transaction-consistent checkpointing (abbreviated as 'Trans' in the figures) resulted in the greatest miss rate. The high miss rate of real-time transactions under the transaction-consistent checkpointing was mainly because of a high data conflict probability, as shown in Figure 9. Under the transaction-consistent checkpointing,

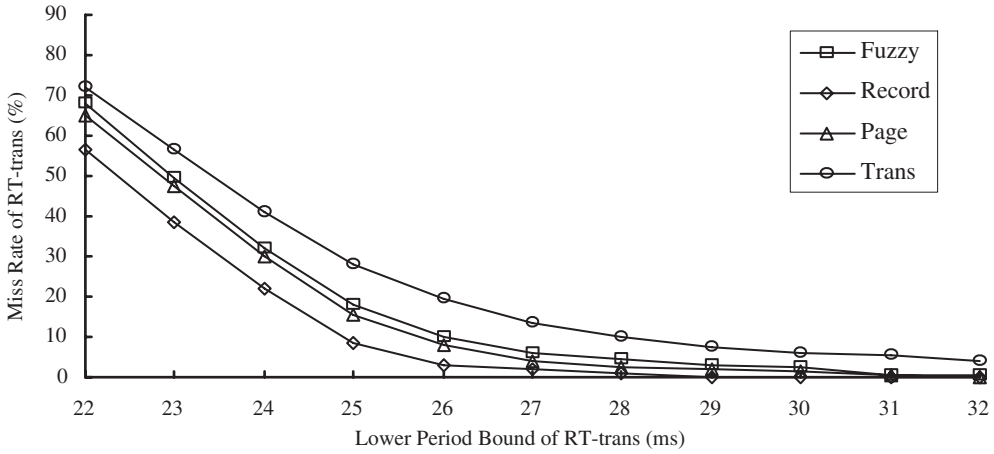

FIGURE 8. The impacts of the lower period bound of real-time transactions on their miss rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 2048 database objects).

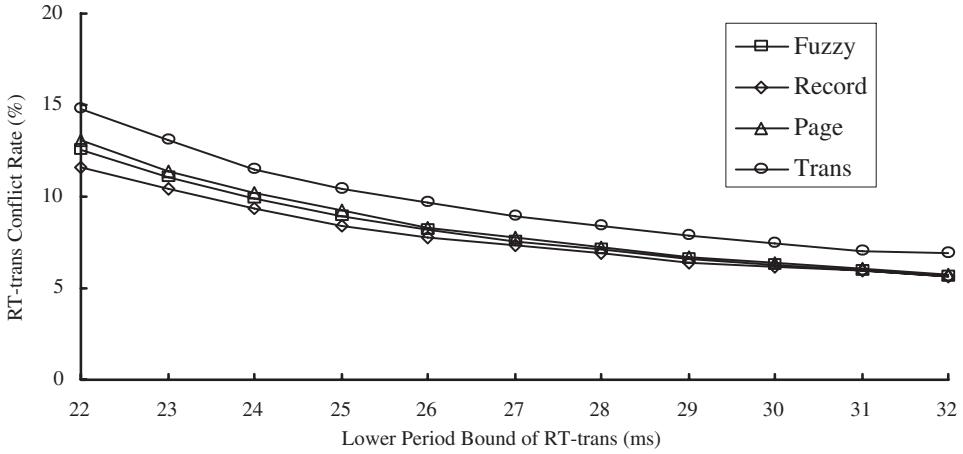

FIGURE 9. The impacts of the lower period bound of real-time transactions on RT-trans conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 2048 database objects).

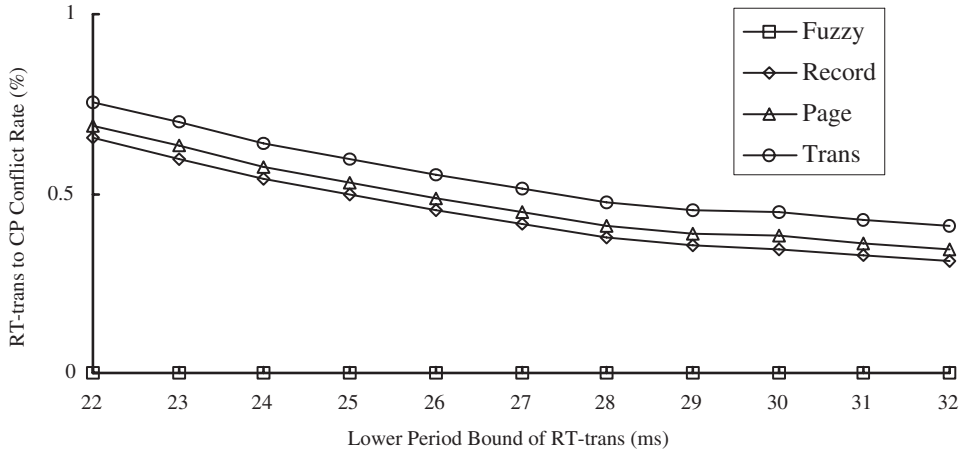

FIGURE 10. The impacts of the lower period bound of real-time transactions on RT-trans to CP conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 2048 database objects).

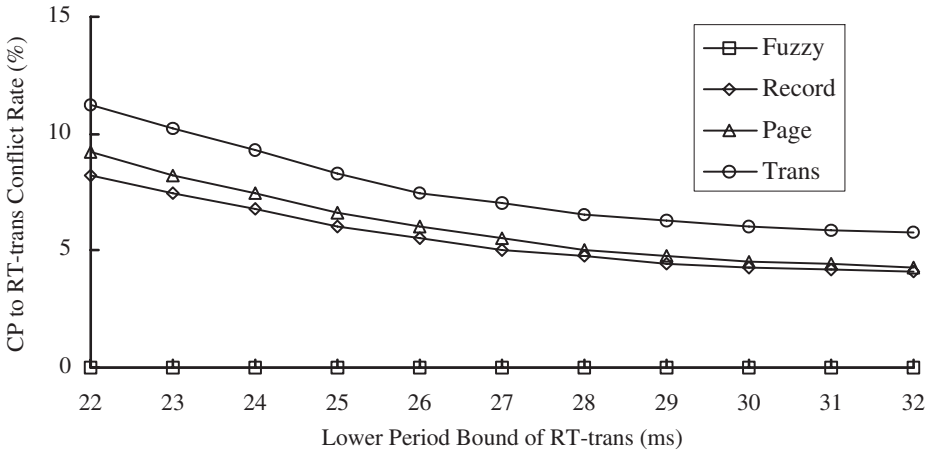

FIGURE 11. The impacts of the lower period bound of real-time transactions on CP to RT-trans conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 2048 database objects).

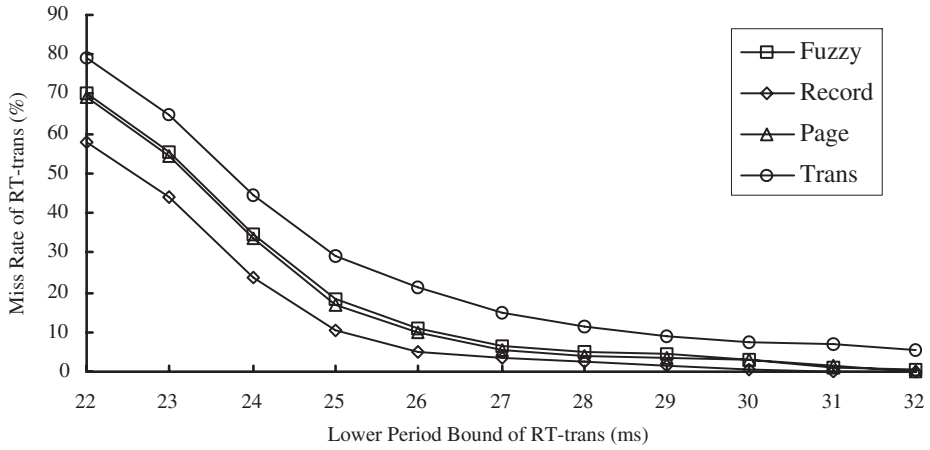

FIGURE 12. The impacts of the lower period bound of real-time transactions on their miss rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the max priority of RT-trans, 2048 database objects).

the checkpointer locked and released their locks in a 2PL fashion. On the other hand, the checkpointer under any other method could release its locks once they were no longer needed for data flushing. As a result, the average lock holding time of the checkpointer under the transaction-consistent checkpointing was much longer than that under any other checkpointing method. More real-time transactions were blocked by the checkpointer, as shown in Figure 10. Note that although the RT-trans to CP conflict rate value was very small (i.e. less than 1%), as shown in Figure 10, its impacts on the executions of real-time transactions were serious since a chain of blocking effects existed. A blocked real-time transaction could induce more blocking later on. Furthermore, the checkpointer, which might block a real-time transaction, might also be blocked by other real-time transactions. As shown in Figure 11, the CP to RT-trans conflict rate under the transaction-consistent checkpointing was also the highest among all checkpointing methods tested in the experiments.

The miss rate of record-based checkpointing was smaller than that of the page-based checkpointing (abbreviated as 'Page' in the figures) due to smaller lock conflict probabilities, as shown in Figure 9, as a result of a smaller checkpointing granule under record-based checkpointing. It was a surprise to see that although fuzzy checkpointing (abbreviated as 'Fuzzy' in the figures) did not introduce any lock conflict with real-time transactions as shown in Figure 10, the miss rate under fuzzy checkpointing was higher than that under record-based checkpointing. The main reasons for this phenomenon were as follows. (1) The checkpointer under fuzzy checkpointing was assigned a priority higher than those of real-time transactions such that the checkpointer was on the winning side in the competition for the CPU. This was also the reason why the RT-trans conflict rate for fuzzy checkpointing was higher than that for record-based checkpointing. (2) The checkpointer did not need to set a lock before it performed the checkpointing method on a data item. Its execution was less affected by the execution of the real-time transactions.

Figures 12–15 provide the performance results when the base priority of the checkpointer was set as the maximum priority of real-time transactions. The corresponding

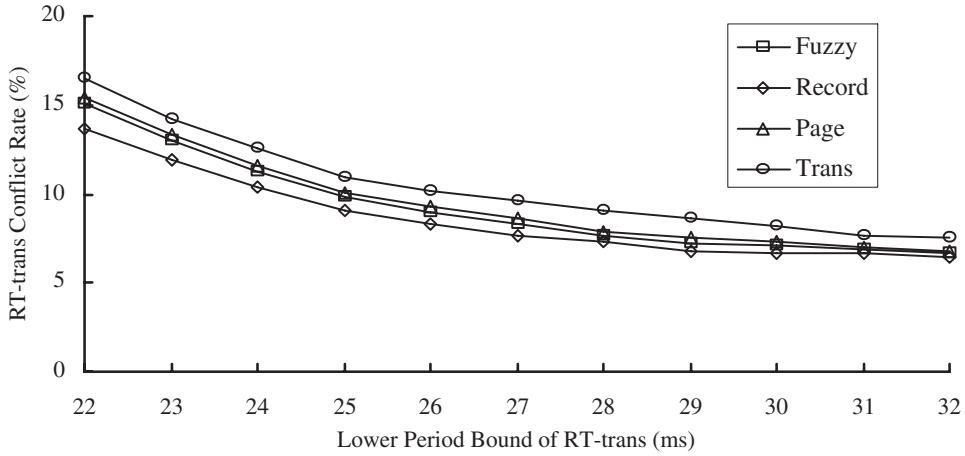

FIGURE 13. The impacts of the lower period bound of real-time transactions on RT-trans conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the max priority of RT-trans, 2048 database objects).

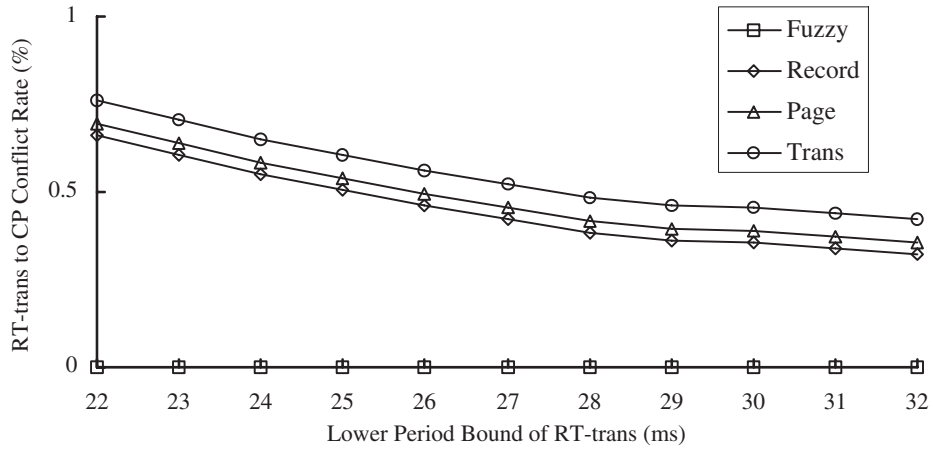

FIGURE 14. The impacts of the lower period bound of real-time transactions on RT-trans to CP conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the max priority of RT-trans, 2048 database objects).

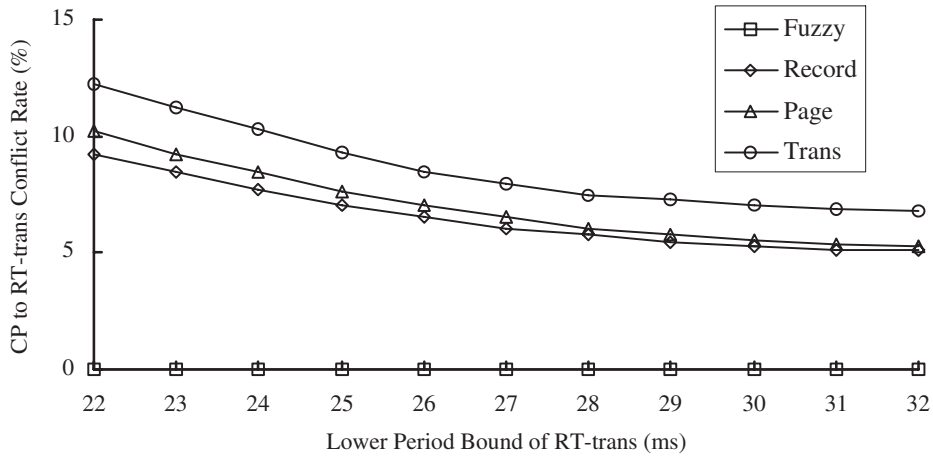

FIGURE 15. The impacts of the lower period bound of real-time transactions on CP to RT-trans conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the max priority of RT-trans, 2048 database objects).

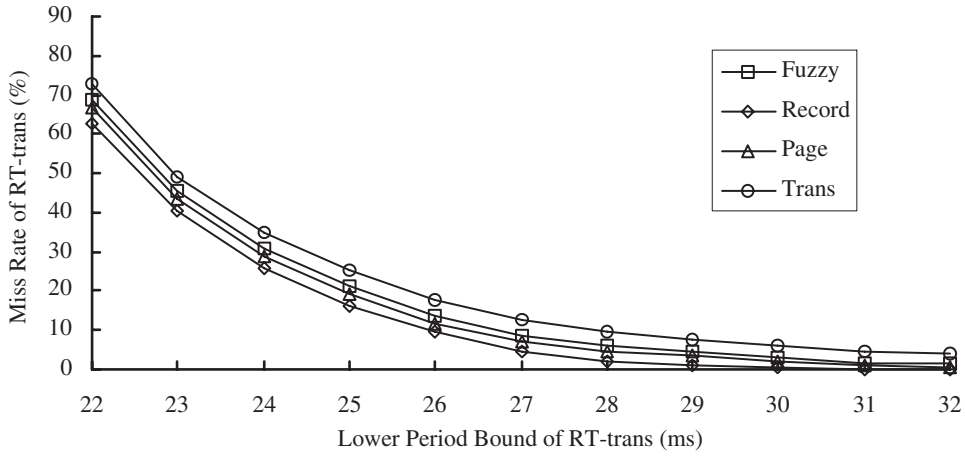

FIGURE 16. The impacts of the lower period bound of real-time transactions on their miss rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 512 database objects).

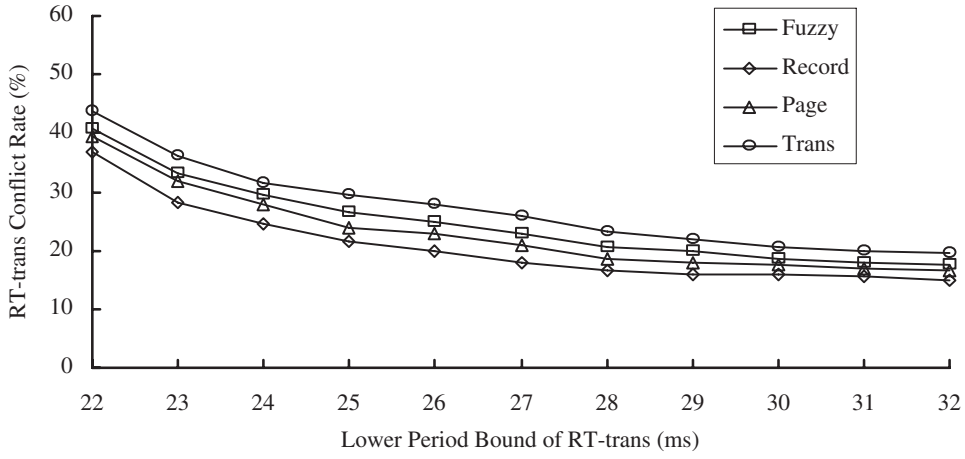

FIGURE 17. The impacts of the lower period bound of real-time transactions on RT-trans conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the mean priority of RT-trans, 512 database objects).

miss rates of the checkpointing methods, as shown in Figure 12, were similar to those shown in Figure 8. However, the miss rates of all the four methods were slightly higher, compared to the results shown in Figure 8, because the impacts of the checkpointing process on the executions of real-time transactions were greater. When we compare Figure 13 with Figure 9, we can see that the increase in the priority of the checkpointer affected not only the executions of real-time transactions but also the conflict rates of real-time transactions. This was because when the checkpointer was assigned a higher priority, real-time transactions faced stronger competition for system resources and, thus, required more time to complete their executions. The lock holding time of real-time transactions tended to be longer.

Figures 16–19 show the performance results when the database only consisted of 512 data objects. The results were close to the expectation: the miss rates of real-time transactions under all of the four methods, as shown in Figures 16 and 18, became higher because of higher conflict probabilities, as shown in Figures 17 and 19. When we compared the results as shown in Figures 8 and 12 with the results as shown in Figures 16 and 18, it was interesting to see that the performance differences amongst the four checkpointing methods became smaller when the database size was smaller. The impacts of raising the base priority of the checkpointer also became less significant. This was because when the database was small, the degree of data contention was already so high that the impacts of the execution of the checkpointer on the executions of real-time transactions became much less significant. The increase of the priority of the checkpointer did not significantly affect the RT-trans conflict rate. This can be observed by comparing the results shown in Figures 17 and 19.

In the experiments, we also varied the period for generating the checkpointer. It was found that the impact of changing the checkpointer period was much smaller when compared with other factors such as the real-time transactions workload and database size, and the relative performance of the checkpointing methods remained the same.

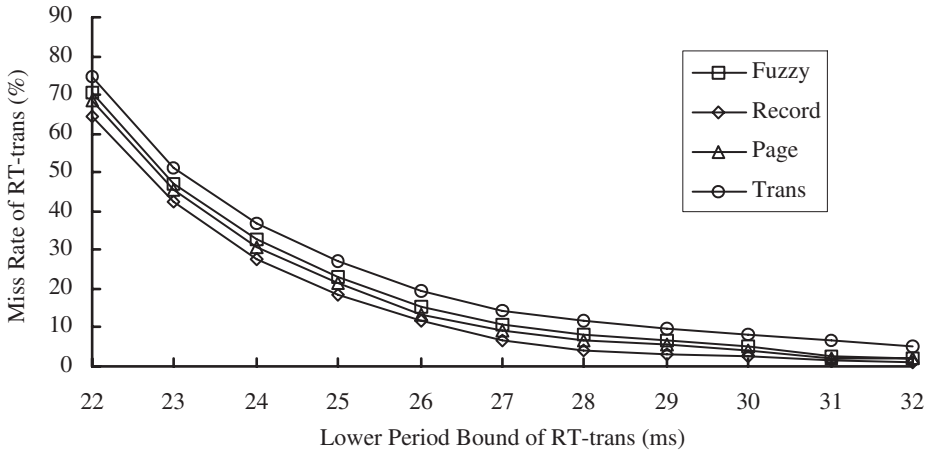

FIGURE 18. The impacts of the lower period bound of real-time transactions on their miss rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the max priority of RT-trans, 512 database objects).

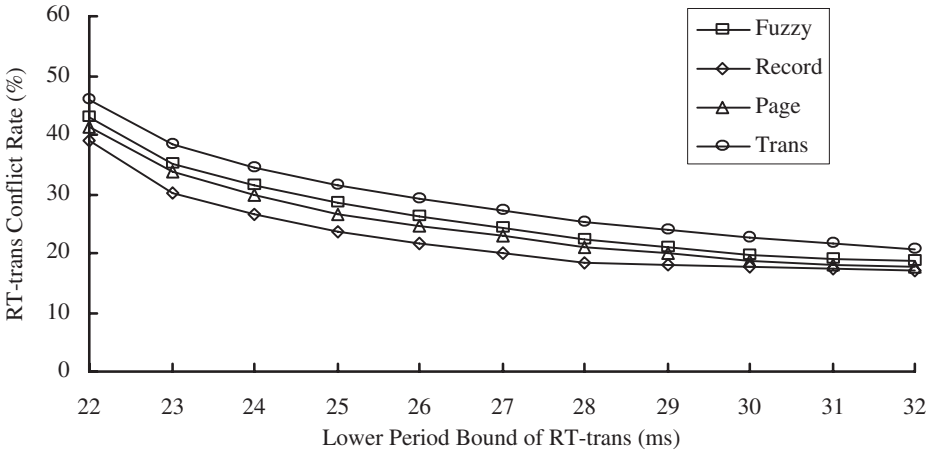

FIGURE 19. The impacts of the lower period bound of real-time transactions on RT-trans conflict rate under RWPCP-DC (the upper period bound of real-time transactions = 45 ms, the base priority of the checkpointer = the max priority of RT-trans, 512 database objects).

7. CONCLUSION

In this paper, we have explored the performance impacts of the durability-guarantee mechanisms on transaction executions in real-time database systems. In particular, we have been interested in the impacts of different checkpointing methods and write-through procedures on lock-based real-time concurrency control protocols. We chose the well-known RWPCP to explore durability and related performance issues. We have shown that schedules generated by the original RWPCP can be irrecoverable and the maximum number of priority inversions for a real-time transaction can be more than one without proper modifications. The performance of RWPCP has been explored when it is revised based on well-known mechanisms such as deferred commitment and extended locking period to maintain the recoverability in transaction execution. We have then modelled the checkpointing process under RWPCP. We have studied the impacts of different checkpointing methods on the performance of the systems. We considered four different checkpointing methods: fuzzy checkpointing, record-based checkpointing, page-based checkpointing and transaction-consistent checkpointing. The four checkpointing methods represented four different locking granularities of data items under checkpointing, which are in a word, a data object, a page or a segment. Priority assignment schemes for the checkpointer were also studied.

A series of simulation experiments was conducted to investigate the cost of recoverability for RWPCP and the impacts of different checkpointing methods on the system performance. It was shown that the extended locking period method could significantly increase the probability of lock conflict among transactions, while a large number of priority inversions introduced by the deferred commitment method could also decrease the system performance. The simulation results showed that the impacts of checkpointing on real-time concurrency control highly depend on the granule size in checkpointing.

In past work, real-time transactions are often assumed to execute in a main-memory database system. Research has rarely addressed the recoverability issues of real-time database systems. For future research, we shall further

explore efficient logging and recovery mechanisms for real-time database systems, especially when slower devices, such as disks, are used for logging. We will also explore the issues related to the 'partial' recovery of a real-time database. Note that in many real-time systems, the capability in bringing the system back to operation quickly is far more important than guaranteeing the perfect precision of the output. Most non-critical data can be either derived or obtained from sensors later, after system recovery. More research on the recovery mechanisms may prove very rewarding in building robust real-time data-intensive applications.

ACKNOWLEDGEMENTS

This work was supported in part by a research grant from the ROC National Science Council under grant No. NSC90-2213-E-002-080 and a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 1078/00E).

REFERENCES

[1] Bestavros, A. (1994) Timeliness via speculation for real-time databases. In *Proc. 15th IEEE Real-Time Systems Symp.*, San Juan, Puerto Rico, December, pp. 36–45.

[2] Dipippo, L. B. C. and Wolfe, V. F. (1993) Object-based semantic real-time concurrency control. In *Proc. 14th IEEE Real-Time Systems Symp.*, New York, December, pp. 87–97.

[3] Kamath, M. U. and Ramamritham, K. (1993) Performance characteristics of epsilon serializability with hierarchical inconsistency bounds. In *Proc. Int. Conf. on Data Engineering*, Vienna, Austria, April, pp. 587–594.

[4] Kuo, T.-W. and Mok, A. K. (1993) SSP: a semantics-based protocol for real-time data access. In *Proc. 14th IEEE Real-Time Systems Symp.*, December, New York, pp. 76–86.

[5] Lee, V. C. S., Lam, K. Y. and Tsang, W. (1998) Transaction processing in wireless distributed real-time databases. In *Proc. 10th Euromicro Workshop on Real-Time Systems*, Spain, January 9–11.

[6] Lin, Y. and Son, S. H. (1990) Concurrency control in real-time databases by dynamic adjustment of serialization order. In *Proc. IEEE 11th Real-Time Systems Symp.*, Orlando, FL, December 4–7.

[7] Ramamritham, K. and Pu, C. (1995) A formal characterization of epsilon serializability. *IEEE Trans. Knowledge Data Eng.*, **7**, 997–1007.

[8] Peng, C.-S. and Lin, K.-J. (1996) A semantic-based concurrency control protocol for real-time transactions. *Proc. IEEE Real-Time Technology and Applications Symp.*, Boston, MA, June 10–12, pp. 59–69.

[9] Sha, L., Rajkumar, R., Son, S. H. and Chang, C. H. (1991) A real-time locking protocol. *IEEE Trans. Comput.*, **40**, 793–800.

[10] Xiong, M., Ramamritham, K., Sivasankaran, R., Stankovic, J. A. and Towsley, D. (1996) Scheduling transactions with temporal constraints: exploiting data semantics. In *Proc. 17th IEEE Real-Time Systems Symp.*, Washington, DC, December, pp. 240–251.

[11] Abbott, R. and Garcia-Molina, H. (1992) Scheduling real-time transactions: a performance evaluation. *ACM Trans. Database Syst.*, **17**, 513–560.

[12] Bernstein, P. A., Hadzilacos, V. and Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA.

[13] Garcia-Molina, H. and Salem, K. (1992) Main memory database systems: an overview. *IEEE Trans. Knowledge Data Eng.*, **4**, 509–516.

[14] Haerder, T. and Reuter, A. (1983) Principles of transaction oriented database recovery. *ACM Comput. Survey*, **15**, 287–317.

[15] Navathe, E. (1994) *Fundamentals of Database Systems* (2nd edn). Addison-Wesley.

[16] Levy, E. and Silberschatz, A. (1992) Incremental recovery in main memory database systems. *IEEE Trans. Comput.*, **4**, 529–540.

[17] Li, X. and Eich, M. H. (1993) Post-crash log processing for fuzzy checkpointing main memory databases. In *Proc. 9th IEEE Int. Conf. on Data Engineering*, Vienna, Austria, April 19–23, pp. 117–124.

[18] Woo, S., Kim, M. H. and Lee, Y. J. (1997) Accommodating logical logging under fuzzy checkpointing in main memory databases. In *Proc. IEEE Int. Database Engineering and Applications Symp.*, Montreal, Canada, August 24–27, pp. 53–62.

[19] Hagmann, R. B. (1986) A crash recovery scheme for a memory-resident database systems. *IEEE Trans. Comput.*, **35**, 839–843.

[20] Salem, K. and Garcia-Molina, H. (1989) Checkpointing memory-resident databases. In *Proc. 5th IEEE Int. Conf. on Data Engineering*, Los Angeles, CA, February, pp. 452–462.

[21] Huang, J. and Gruenwald, L. (1996) An update-frequency-valid-interval partition checkpoint technique for real-time main memory databases. In *Proc. First Int. Workshop on Real-Time Databases: Issues and Applications*, Newport Beach, CA, March, pp. 135–143.

[22] Sivasankaran, R. M., Ramaritham, K. and Stankovic, J. A. (2001) System failure and recovery. *Real-Time Databases Systems*, pp. 109–124.

[23] Ramamritham, K. (1993) Real-time databases. *Int. J. Distrib. Parallel Databases*, **1**, 199–226.

[24] Huang, J., Stankovic, J. and Ramamritham, K. (1992) Priority inheritance in soft real-time databases. *J. Real-Time Syst.*, **4**, 243–268.

[25] Haritsa, J. R., Carey, M. J. and Livny, M. (1990) On being optimistic about real-time constraints. In *Proc. 9th ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems*, Nashville, TN, April, pp. 331–343.

[26] Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990) Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.*, **39**, 1175–1185.

[27] Lee, D. and Cho, H. (1997) Checkpointing schemes for fast restart in main memory database systems. In *1997 IEEE Pacific Rim Conf. on Communications, Computers, and Signal Processing*, Victory, BC, July 7, pp. 663–668.

[28] Jagadish, H. V., Silberschatz, A. and Sudarshan, S. (1993) Recovering from main memory lapse. In *Proc. 19th VLDB Conf.*, Dublin, Ireland, August, pp. 391–404.

[29] Gupta, R., Haritsa, J. and Ramamritham, K. (1997) More optimism about real-time distributed commit processing. In *Proc. 19th IEEE Real-Time Systems Symp.*, Madrid, Spain, December 2–4.

[30] Kim, Y. K. and Son, S. H. (1996) Supporting predictability in real-time database systems. In *Proc. Real-Time Technology*

and Applications Symp., Brookine, MA, June 10–12, pp. 38–48.

[31] Locke, C. D., Vogel, D. R. and Mesler, T. J. (1991) Building a predictable avionics platform in Ada: a case study. In *IEEE 12th Real-Time Systems Symp.*, San Antonio, TX, December 4–7, pp. 181–189.

[32] Ulusoy, O. and Buchmann, A. (1998) A real-time concurrency control protocol for main-memory database systems. *Inform. Syst.*, **23**, 109–125.

[33] Intel Value Series 100 Flash Memory Card. http://developer.intel.com/design/fcard/prodbref/29768602.htm, 1998.

[34] Burns, A., Wellings, A. J., Bailey, C. M. and Fyfe, E. (1993) The Olympus Altitude and Orbital Control System: a case study in hard real-time system design and implementation. In Gauthier, M. (ed.), *Ada sans frontieres. Proc. 12th Ada-Europe Conf.*, Lecture Notes in Computer Science, **688**, 19–35. Springer-Verlag.