An Update-Frequency-Valid-Interval Partition Checkpoint Technique for Real-Time Main Memory Databases[†]

Jing Huang Le Gruenwald

School of Computer Science
University of Oklahoma
Norman, OK 73019

Abstract In this paper, we propose a checkpoint technique for real-time main memory database (MMDB) systems which aims at not only reducing the system recovery time, but also increasing the percent of transactions meeting their deadlines and enabling as many temporal data as possible to be used before losing their validity. The main idea of this approach is to divide the MMDB into partitions based on data types (persistent vs. temporal), update frequencies and temporal valid intervals, and checkpoint each partition independently based on its update frequency and temporal valid interval. Our simulation results show that the proposed scheme offers a significant performance improvement over the conventional fuzzy checkpoint approach, and the key factors in the design of a real-time MMDB checkpoint technique include how the MMDB is partitioned, whether temporal data of short valid intervals are checkpointed and how the checkpoints among partitions are scheduled.

1. Introduction

A real-time database system (RTDBS) is the one in which transactions must not only maintain the consistency constraints of the database but also satisfy their timing constraints (such as deadlines). In addition to transaction deadlines, an RTDBS often involves processing both temporal data which lose their validity after a certain period of time, as well as persistent data which remain valid regardless of time ([13], [14]). The main goal of an RTDBS is to meet the timing constraints of transactions and data.

During the last few years, a great number of researchers have been attracted to real-time database systems, and many related areas such as transaction scheduling, concurrency control and I/O scheduling have been extensively studied. However, little research has been done on checkpointing. *Checkpointing* is a process used to maintain on disks an up-to-date copy of the database and thereby provides a starting point for log recovery. When a system failure occurs, as checkpoints provide an almost up-to-date copy of the database, most data in the log are not needed. The recovery procedure only needs to process the log records generated after the last complete checkpoint. The efficiency of the checkpoint technique therefore has an important effect on the performance of database recovery, especially in a *main memory database* (MMDB) environment in which all or a major portion of the database can be memory-resident. As I/O activities are kept at minimum, transaction processing can be processed with little interference. It is obvious that the use of MMDBs has a potential for obtaining a substantial performance improvement over a disk-resident database system for real-time applications. However, due to the timing constraints imposed on both transactions and data in real-time systems, and the vulnerability to failures of MMDBs, checkpoint issues need to be re-explored or re-designed in such a new environment.

In this paper, an *update-frequency-valid-interval partition checkpoint* (UFVIPC) scheme is proposed for real-time MMDB system. Simulation experiments are conducted to measure the performance of the proposed checkpoint scheme. The remainder of this paper is organized as follows. Section 2 describes the media organization required by the UFVIPC scheme. Section 3 gives a detailed description of the proposed scheme. The simulation model and methodology are introduced in Section 4. Performance experiments and results are presented in Section 5. Lastly, Section 6 concludes the paper.

2. Media Organization Required by the UFVIPC Scheme

Before describing the UFVIPC scheme, this section introduces the media organization required by

[†] This material is based in part upon work supported by National Science Foundation under Grant No. IRI-9201596.

the proposed partition checkpoint scheme. As shown in Figure 1, an MMDB, which resides in the volatile main memory (MM), is divided into partitions and each of which consists of a page or a set of pages. Temporal data are partitioned based on their temporal valid intervals and persistent data are partitioned based on their update frequencies. Pages which belong to one partition are not necessarily placed next to each other in MM, but are linked together so that they can be located easily during the checkpoint process. In the UFVIPC scheme, it is assumed that persistent data pages are linked together in MM according to the decreasing order of update frequencies while temporal data pages are linked in MM based on the increasing order of valid intervals. Partitions may or may not be of the same size. Note that in Figure 1, $v(P_i)$ stands for the valid interval value of page P_i and $f(P_i)$ the update frequency of page P_i.

Due to the volatility of semiconductor memory, the backup copy of the MMDB is kept in an archive memory (AM) residing on secondary storage. The database on AM is updated only when a checkpoint is taken. Fuzzy checkpointing [5], which copies the database from MM to AM periodically, is assumed in this work. As this technique does not require the system to be quiescent during its execution, it offers a higher system performance for a real-time MMDB system than consistent checkpointing does.

In order to facilitate database recovery, each partition is associated with a local checkpoint bit map and a local log buffer residing on the nonvolatile memory. Local checkpoint bit maps are used to assist the implementation of fuzzy checkpoint. A bit in a local checkpoint bit map corresponds to a page in its associated partition and indicates whether the page has been modified (or dirty) since the last local checkpoint. Note that a checkpoint performed on a partition is referred to as local checkpoint. Only the modified pages will be flushed out from MM to AM during the checkpoint process.

Log records generated during normal processing are grouped according to partitions and stored in the corresponding local log buffers. Based on our previous studies [4], when being combined with deferred update, the logging scheme, which logs both valid and invalid temporal data, and maintains persistent data and temporal data log records separately, gives the best performance in terms of logging space, number of memory references and cost to perform log processing. This technique is thus used to handle logging in this work. We assume that log buffers are large enough to contain all updates of active transactions. When a log buffer is full, its contents will be flushed to a log disk. One benefit of using local log buffers is that they permit partitions to be recovered independently after a system crash, thus transactions can start executing before all partitions are recovered, and partitions can be recovered on demand.

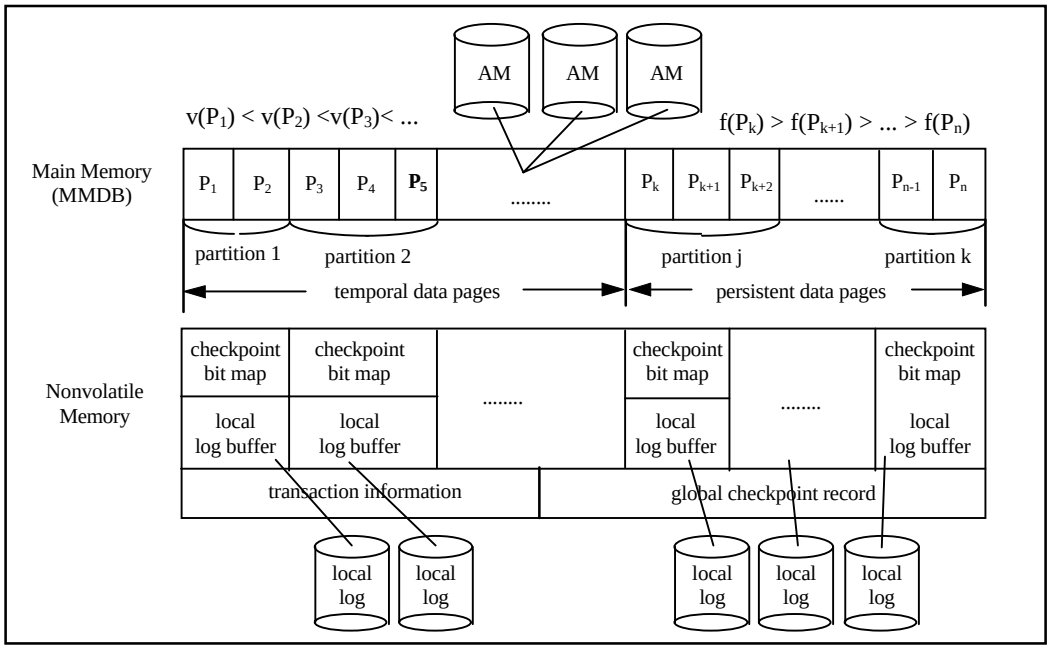

Figure 1. Media Organization Required by the UFVIPC Scheme

In order to fasten the recovery process, a transaction abort list is maintained in the non-volatile memory during normal processing. At the time of database recovery, by making use of this table, the fate of each transaction in the log can be decided. The global checkpoint record is used to record the location of the last complete local begin checkpoint record for each partition, which provides the start point for post-crash log processing. The global checkpoint record is updated when each checkpoint invoked on a partition is completed.

Two processors, Database Processor (DP) and a Recovery Processor (RP), are assumed in the system and they are running in parallel. The DP handles normal transaction execution while the RP manages transaction termination, logging, checkpointing and recovery from a system failure.

3. Update-Frequency-Valid-Interval Partition Checkpoint (UFVIPC) Scheme

The motivation of the UFVIPC scheme is to take advantage of hot spots [12] (parts of the database that are accessed frequently) and make use of the timing constraint information associated with data so that the checkpoint performance can be improved. The main features of the UFVIPC algorithm include 1) temporal data are partitioned based on valid intervals while persistent data are partitioned based on update frequencies; 2) partitions with higher update frequencies (hot partitions) are checkpointed more often than those of lower update frequencies so that the log information that needs to be processed for hot partitions can be reduced and the overall recovery time is minimized; 3) temporal data partitions with short valid intervals are given more opportunities to be flushed out by increasing their update frequencies than other partitions, this enables many temporal data to be recovered and used before losing their validity and in turn reduces the number of transaction aborted or delayed due to invalid data access; and 4) no logging and checkpointing will be invoked on temporal data whose valid intervals are shorter than the specified interval threshold, this not only reduces the interference of logging and checkpointing activities on normal operations but also hastens the recovery process.

The idea of partition checkpoint was also proposed in ([8], [9]); however, not like ours, neither transaction deadlines nor temporal consistency requirements were considered in their schemes. In the following subsections, we first introduce how the update frequency of temporal data partitions is increased, how the checkpoint frequency of a partition is assigned and then give the detailed description of the UFVIPC scheme.

3.1. Increasing Update Frequencies for Temporal Data Partitions

In order to enable temporal data of short valid intervals to be checkpointed more frequently than persistent data partitions, the UFVIPC scheme increases update frequency of each temporal data partition *i* by using the following formula:

$$UFi = AUFi * \left\lceil (Lt - LP + 1) \Big/ \left\lfloor \frac{Vi}{Vthreshold} \right\rfloor \right\rceil$$

where UF_i represents the increased update frequency of the partition *i*, AUF_i the actual update frequency of a temporal data partition *i*, which is the sum of update frequencies of all pages in partition *i*, V_i the valid interval of the temporal data partition *i*, which is the shortest valid interval among all pages in partition *i* and $V_{threshold}$ the interval threshold selected by the system, L_t and L_p the log record size of temporal data and persistent data, respectively. Note that $L_t > L_p$ and $V_i \geq V_{threshold}$. always hold in the above formula, the item in the ceiling function specifies the amount by which the actual update frequency of each temporal data partition is increased. This amount is decreased from (L_t-L_p+1) to 1 as the temporal valid interval V_i of partition *i* is increased. The shorter temporal valid interval a partition has, the larger increase the partition's update frequency is given. This is because the shorter a partition's temporal valid interval is, the more frequently the partition will be updated, the more log records will be accumulated in the corresponding local log buffer. It is therefore desired that partitions with short temporal valid intervals are checkpointed as frequently as possible so that less log information needs to be examined for recovery. However, in order to avoid persistent data partitions being ignored for a long time during the checkpoint process, the amount of increase in update frequencies of temporal data partitions is decreased as the temporal valid interval of a partition is increased. Since, otherwise, too many log records will be accumulated on persistent data partitions, the overall performance may be degraded. The detailed explanations of the above formula can be found in [7].

3.2. Assigning Checkpoint Frequency

In order to enable partitions with a high update frequency to be checkpointed more frequently, the checkpoint frequency of each partition is assigned in such a way that the higher the update frequency of a partition is, the higher checkpoint frequency the partition has. Let CF_i be the checkpoint frequency of a partition i, and N_{part} be the number of partitions the database has. The CF_i of the partition i can be decided based on its UF_i as follows:

$$\mathrm{CFi} = \left\lceil \left(\mathrm{UFi} \Big/ \sum_{i=1}^{Npart} UFi \right) * \mathrm{Npart} \right\rceil$$

The objective of the above formula is to guarantee that if $UF_i > UF_j$, then $CF_i > CF_j$. Besides, the value of CF_i should reflect the relative update frequency of partition i and enable each partition to be checkpointed at least once in a limited period of time. The first item, $\mathrm{UFi} \Big/ \sum_{i=1}^{Npart} UFi$, represents the relative update frequency of a partition i among all partitions. The second item, N_{part}, is used to calculate the checkpoint frequency of the corresponding partition. Note that if CF_i is assigned to be UF_i, although it can reflect the relative update frequency order among partitions, as UF_i is usually a large number, it is not easy to schedule checkpoints so that frequently updated partitions can be given more attention and at the same time other partitions are not ignored for a too long time. The ceiling function used here guarantees that each partition will be checkpointed at least once. This avoids the starvation problem in which some partitions may never have a chance to be checkpointed.

3.3. The Algorithm

In detail, the UFVIPC algorithm consists of the following steps:

1. Partition persistent data based on their update frequencies.
2. Partition temporal data whose valid intervals are longer than the interval threshold based on their valid intervals. For temporal data of valid intervals shorter than the interval threshold, no logging and checkpointing process will be incurred on them.
3. Let $\mathrm{AUF_i}$ and $\mathrm{UF_i}$ be the actual update frequency and the increased update frequency of each partition, respectively. For a persistent data partition j, $UF_j = AUF_j$. For a temporal data partition i, the UF_i is calculated by using the following formula: $UFi = AUFi * \left\lceil (\mathrm{Lt} - \mathrm{Lp} + 1) \Big/ \left\lfloor \frac{Vi}{Vthreshold} \right\rfloor \right\rceil$.
4. Set the checkpoint frequency $\mathrm{CF_i}$ of each partition i ($1 \le i \le Npart$) based on its update frequency $\mathrm{UF_i}$ as follows: $\mathrm{CFi} = \left\lceil \left(\mathrm{UFi} \Big/ \sum_{i=1}^{Npart} UFi \right) * \mathrm{Npart} \right\rceil$.
5. Find a partition i which has the highest checkpoint frequency among $\mathrm{N_{part}}$ partitions, that is
   CFi >= CFj, j =1, 2, ... , i-1, i+1, ..., $\mathrm{N_{part}}$
6. Perform a local checkpoint on partition i.
7. Set CFi = CFi -1.
8. If there exists a partition i such that $\mathrm{CF_i} > 0$, goto Step 5; Otherwise, goto step 9.
9. Reset the checkpoint frequencies $\mathrm{CF_i}$'s for all partitions i's to their original checkpoint frequencies and goto Step 5.

Note that Step 6 invokes a local checkpoint on partition i, which is performed as follows:

1. write a BC (Begin Checkpoint) record into the local log buffer of partition i.
2. save the location of the BC record in the global checkpoint record
3. ***While*** there is an unchecked page in the partition i ***Do***
   If the page is dirty ***Then***
   reset the corresponding bit in the local checkpoint bit map of partition i to 0.
   copy the page to the IO buffer.
   request IO to flush this page to the AM
   end of ***If***
   end of ***While***
4. write an EC (End Checkpoint) record in the local log buffer of partition i.
5. record the location of the last complete local checkpoint in the global checkpoint record

As transaction execution cannot proceed until requested pages are brought into MM, interval threshold can be decided based on reload granularity, which is the smallest unit of data to be reloaded without preemption used by the system at recovery time. As a cylinder is selected to be the reload granularity in this study, the value of interval threshold is defined to be the time needed to reload an entire cylinder from AM to MM.

4. Simulation Model and Methodology

In order to measure the performance of the proposed checkpoint algorithm, we developed a simulation model of a centralized real-time MMDB system and performed extensive experiments. Only firm deadline transactions, which are discarded if they are not completed by their deadlines, are considered in the model. The *earliest deadline policy* [1] is selected for transaction execution priority assignment. This policy gives a transaction that has the earliest deadline the highest execution priority. If two transactions have the same deadline, the transaction which arrives at the system earlier is considered to have a higher execution priority. The "conditional restart" using the 2-phase locking concurrency control mechanism [1] is adopted in our simulation. For simplicity, we assume that when a transaction enters the system, before it can be processed, it must obtain all needed locks on pages it is going to access. At its commit time, the transaction releases all its locks.

The parameters used to specify the system configuration and workload are summarized in Tables 1 and 2. All the parameter values are derived based on the DEC 3000 Model 400/400S AXP Alpha workstations [4] and Micropolis 22000 disk drivers [10], since they accommodate high performance applications. The parameter *db_size* corresponds to the number of data pages stored in the database. Transaction arrivals are assumed to be exponentially distributed with the mean value of *arrival_rate*. To prevent the possibility of transaction overload in the system, the total number of active transactions in the system is limited by the parameter *max_mpl* [15]. The number of data objects accessed by a transaction is determined by the parameter *trans_size*, which is uniformly distributed between 5 and 20 [2]. Page accesses are exponentially distributed across the whole database. For each data access of a normal transaction, the probability that the accessed data object will be updated is determined by the parameter *prob_write* [15]. The type of the accessed data object is decided based on the parameter *per_temporal*. Temporal data valid intervals are uniformly distributed within the range specified by *min_interval* and *max_interval*. *slack_factor* is used in assigning new transaction deadlines [15].

The CPU power, memory access time and word size are chosen based on the DEC 3000 model machine [4]. The parameters *pre_trans* and *pre_op* denote the CPU costs to preprocess a transaction (e.g., allocating and initializing needed lock tables, logging "begin transaction") and an operation (e.g., fetching one instruction), respectively. *sm_access* indicates the time required to perform a read or write on the non-volatile memory or stable memory (SM). In [4], SM access time is assumed to be 10% slower than memory access time to account for the possible use of BBRAM to make the SM nonvolatile. Disk parameters are chosen based on 2200 series-SCSI Micropolis disks [10]. The average disk transfer time is 0.128 milliseconds for 1024 bytes. Since the 2-phase locking concurrency control protocol is used and each transaction performs operations on pages, a page-level lock granularity is chosen. The time to lock and unlock operations are represented by parameters *lock_time* and *unlock_time*, which together with the time to allocate or release a main memory page, and the time to request a disk I/O is estimated based on the number of instructions that are needed in order to perform the corresponding task and DEC 3000 model machines. As a cylinder is assumed to be the reload granularity for prefetch reload, interval threshold is therefore chosen to be the time required to reload the entire cylinder.

Based on whether transactions can update temporal data, we classify transactions into two groups: aperiodic transactions and periodic transactions. *Aperiodic transactions* or *normal transactions* are generated using an exponential distribution stream at a specified mean rate. Each normal transaction submitted to the system is associated with its creation time, transaction identifier, transaction size, operations, pages on which operations are performed, deadline and execution priority. *Periodic transactions* are write-only transactions. Each periodic transaction is responsible for updating one temporal data page and is invoked at the beginning of its update period. The time interval during which a periodic transaction is triggered in order to maintain the absolute temporal consistency is called update period. The update period of a periodic transaction is defined to be half of the corresponding temporal data's valid interval. The deadline of a periodic transaction is assumed to be the end of its update period.

If a normal transaction is found to read an invalid temporal data page, in order to give a chance to the periodic transaction which is responsible for updating the temporal data page, the normal transaction will be aborted and release all its locks. If the normal transaction still has a feasible deadline, it will be scheduled to restart later; otherwise, it is discarded.

The performance metric used here is the percent of transactions missing deadlines, which is observed from the beginning of a simulation run until the end

of the simulation. The conventional fuzzy checkpoint approach is chosen as a comparison.

Except in the testing case which examines the effects of the system failure rate on the proposed checkpoint scheme, in all other testing cases a system crash is assumed to take place once at the time when half of the specified normal transactions are finished (committed or aborted). When a system failure occurs, all resources are made inactive, and all active transactions are kept in a file so that they can be restarted later. The MM is emptied to simulate its volatility. Transaction execution is resumed when the entire database is reloaded in MM and recovered. The simulation model is written using the simulation language SLAM II [11]. In each simulation experiment at least 20,000 normal transactions are executed. The final results are obtained by averaging the results over 20 independent runs. 95% confidence levels are obtained for the performance results. The width of the confidence interval of each data point is within 5% of the point estimate.

Parameter	Meaning	Default Value	Range
db_size	database size	900 pages	600 --- 2100
prob_write	probability of write	40%	0% --- 100%
prob_read	probability of read	60%	100% - prob_write
num_parts	number of partitions	10	2 --- 15
per_temporal	percent of temporal data	40%	0% --- 100%
per_persist	percent of persistent data	60%	100% - per_temporal
min_interval	minimum temporal valid interval	40 ms	40 --- 3000
max_interval	maximum temporal valid interval	1500 ms	40 --- 3000
slack_factor	slack factor	5	5 --- 20
arrival_rate	transaction arrival rate	200 transactions/second	50 --- 350

Table 1. Dynamic Parameters

Parameter	Meaning	Default Value
max_mpl	multiple programming	10
trans_size	transaction size	10 operations
pg_size	page size	23476 bytes
alloc/releas_tm	allocate/release a main memory page	0.005 ms
am_req_tm	request a write/read to/from archive disk	0.00143 ms
pre_trans	preprocess 1 transaction	0.0072 ms
pre_op	preprocess 1 operation	0.000007 ms
bmap_tm	read bit map time	0.02957 ms
sm_access	stable memory access time per word	0.000198 ms
et_tm	end transaction	0.0054 ms
mm_access	main memory access time per word	0.00018 ms
int_io_tm	initialize log I/O time	0.0014 ms
log_io_tm	write a log page to disk	5.624
word_sz	number of bytes per word	8
per_word	words per persistent data log record	4
tem_word	words per temporal data log record	6
seek_tm	average seek time	10 ms
latency_tm	average latency time	5.56 ms
transfer_tm	time to transfer 1 data page	0.064 ms
am_disk	number of AM disks	2
log_pg_sz	log page size	2000 bytes
lock_tm	get one lock (one try)	0.0007 ms
unlock_tm	release one lock	0.0007 ms
interval_threshold	interval threshold is the time required to reload the entire cylinder	97 ms
cpu_power	CPU(DP and RP) power	140 MIPS

Table 2. Static Parameters

5. Simulation Results

Based on the simulation results obtained, the following conclusions are drawn: the UFVIPC scheme offers better overall performance than the conventional fuzzy checkpoint approach does. The key factors (summarized in Table 3) that affect the performance of the proposed partition checkpoint scheme are how many partitions the database has, whether the temporal data of short valid intervals is checkpointed and how the checkpoints among partitions are scheduled. Our results show that the more partitions the database has, the more performance improvement the system tends to obtain (shown in Figure 2, where *UFVIPCi* means an MMDB is divided into *i* partitions). However, UFVIPC does impose an overhead on normal system operations, and the more partitions the database has, the more overhead it incurs. When the number of partitions reaches a certain limit, the benefit obtained from further partitioning is not very significant. Not logging and checkpointing temporal data whose valid intervals are shorter than the interval threshold does help to improve the overall performance. This is especially true when most of temporal data valid intervals are short as plotted in Figure 3 and the system has a great amount of temporal data. The results also indicate that giving temporal data partitions, especially those of short valid intervals, more chances to be flushed to AM is desired, which reduces the amount of log information that needs to be processed at recovery time.

The performance improvement offered by UFVIPC is lessened when the transaction arrival rate or the system failure rate increases, as shown in Figures 4 and 5, respectively. This is because as the transaction arrival rate or system failure rate gets higher, the number of transactions that are backlogged during a system crash increases tremendously. The savings obtained from reducing post-crash log processing time, therefore, becomes less significant compared to those obtained in a system of a low system load and a low failure rate.

Design factor	Importance	Expected value	Reason
number of partitions	yes	high	• less time is needed to finish a local checkpoint • less log information needs to be examined for recovery
not logging and checkpointing temporal data of short valid intervals	yes	short temporal valid interval	• reduce logging and checkpointing overhead • reduce overhead imposed on normal processing • less log information needs to be processed for recovery
schedule checkpoints among partitions	yes	based on different priorities	checkpoint temporal data of short valid intervals more often so that total amount of log information can be reduced for recovery

Table 3. Key Factors in the Design of a Real-Time MMDB Checkpointing Technique

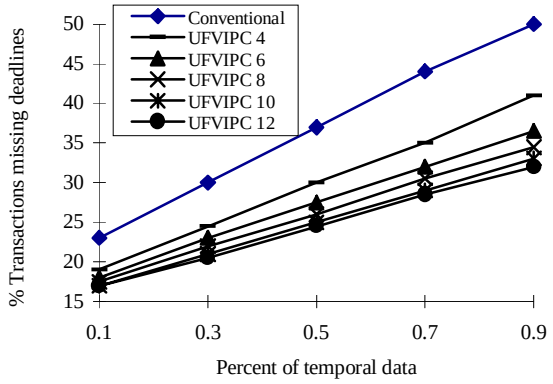

Figure 2. Percent of temporal data vs. % transactions missing deadlines

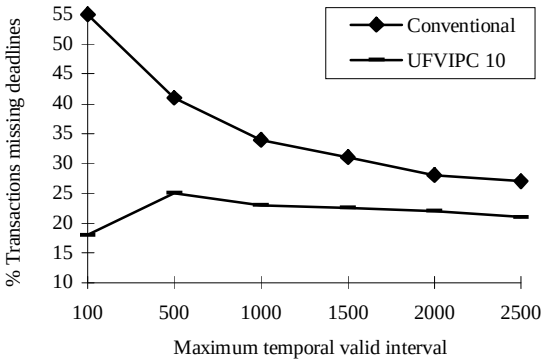

Figure 3. Maximum valid interval vs. % transactions missing deadlines

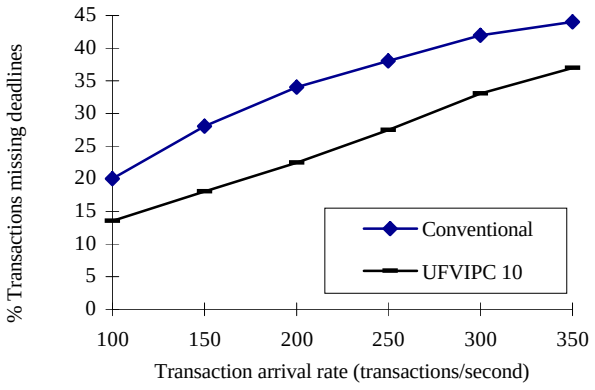

Figure 4. Transaction arrival rate vs. % transactions missing deadlines

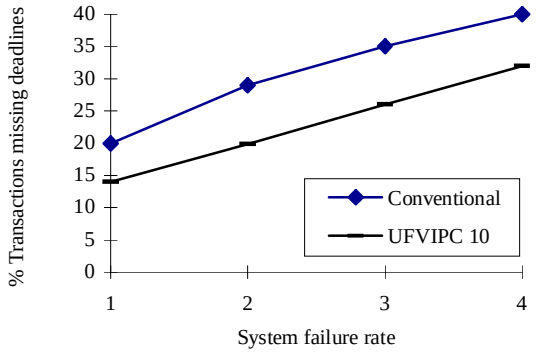

Figure 5. System failure rate vs. % transactions missing deadlines

6. Conclusions

In this paper, an update-frequency-valid-interval partition checkpoint (UFVIPC) is proposed for real-time MMDBs, which aims at not only reducing the post-crash log processing time but also enabling many temporal data to be used before losing their validity. The main features of the UFVIPC scheme include 1) persistent data are partitioned based on update frequencies while temporal data are partitioned based on valid intervals; 2) both data timing constraints information and update frequencies are taken into account during the scheduling of checkpoints; 3) temporal data partitions, especially those of short valid intervals, are given more opportunities to be flushed to AM disks than persistent data partitions; and 4) temporal data objects whose valid intervals are shorter than a specified interval threshold are not logged and checkpointed during the normal operation. The simulation results show that the proposed scheme outperforms the conventional fuzzy checkpoint approach.

References

[1] R. Abbott, H. Garcia-Molina, “Scheduling Real-Time Transactions: A Performance Evaluation”, ACM Transaction on Database Systems, Vol. 19, No. 3, Sept. 1992, pp. 513-560.

[2] S. Chakravarthy, D. Hong and T. Johnson, “Real-Time Transaction Scheduling: A Framework for Synthesizing Static and Dynamic Factors”, University of Florida, Computer and Information sciences, Technical Report, March 14, 1994.

[3] C. H. Corti, M. H. Eich, “Update and Logging Options in Main Memory Database”, Technical Report 90-CSE-13, Department of Computer Science and Engineering, South Methodist University, March 1990.

[4] DECdirect Workgroup Solutions Catalog, Winter 1993.

[5] R. B. Hagmann, "A Crash Recovery Scheme for a Memory-Resident Database System", IEEE Transactions on Computers, Vol. C-35, No. 9, September. 1986.

[6] J. Huang, L. Gruenwald, “Logging Real-Time Main Memory Databases”, Proceedings of International Computer Symposium, December 1994, pp. 1291-1296.

[7] J. Huang, “Recovery Techniques in Real-Time Main Memory Databases”, Ph. D. Dissertation, School of Computer Science, the University Oklahoma, 1995.

[8] H. V. Jagadish, A. Silberschatz, S. Sudarshan, “Recovering From Main Memory Lapse”, Proceedings of the 19th VLDB Conference, 1993, pp. 391-404.

[9] Xi. Li, M. H. Eich, “Partition Checkpointing in Main Memory Database”, Technical Report 93-CSE-23, Department of Computer Science and Engineering, South Methodist University, Dallas.

[10] 22000 Series - SCSI Micropolis Disk Drive Information, 1993.

[11] A. Alen B. Pritsker, “Introduction of Simulation and SLAM II”, John Wiley & Sons, Inc., New York, 1986.

[12] K. Salem, D. Barbara, “Probabilistic Diagnosis of Hot Spots”, IEEE International Conference on Data Engineering, 1992, pp. 30-39.

[13] R. M. Sivasankaran, K. Ramamritham, J. A. Stankovic , D. Towsley, “Data Placement, Logging and Recovery in Real-Time Active Databases”, International Workshop on Active and Real-Time Database Systems (ARTDB-95), June 9-11, 1995.

[14] S. H. Son, Y. Kim, “Predictability and Consistency in Real-Time Database Systems”, Proceeding of Information Science, 1993.

[15] O. Ulusoy, G. Belford, “Real-Time Transaction Scheduling in Database Systems”, Information Systems, Vol. 18, No. 8, 1993, pp. 559-580.