Efficient Durability Support for Multicore In-Memory Database

Hao Qian[1, a]

[1]*School of Software, Shanghai Jiao Tong University, 200240, China*

[a]*qh0315@126.com*

Abstract

The increasing core counts and memory volume open opportunities to perform online transactional processing (OLTP) in memory, leading to a new type of databases, called in-memory databases. However, this also makes durability and failure recovery a critical issue due to the volatility of main memory. In this paper, we describe the design and implementation of Gawain, which provides both efficient logging and fast recovery for in-memory databases.

Keywords: Multicore; In-memory Database; Durability; Recovery

1 Introduction

The drastic increases of core counts and memory volume and emerging workloads demanding high throughput and low latency have led to a new kind of databases, namely in-memory databases. Examples of in-memory databases include Silo [1], DBX [2] and Hyper [3].

While processing data in memory leads to significant performance boost, it also faces a serious challenge: durability. As all database records are stored in volatile memory during processing, a machine crash or power outage may not only render hours of work useless, but also cause loss of critical customers' data.

This paper tries to bridge the gap between high-performance in-memory computing and the required interaction of durability with slow disks. To this end, this paper presents Gawain, a set of techniques to provide efficient durability support for high-performance in-memory databases. The goal of Gawain is incorporating efficient logging and recovery into in-memory databases, yet incurring very small overhead to latency and throughput.

We have implemented Gawain for DBX. Performance evaluations on a 4-core 8-thread machine show that Gawain incurs abort 10% overhead compared to purely memory-based version and the parallel recovery allows recovery of 8 gigabytes data in just 4.75 seconds. The parallelization of recovery reduces the time from 17.38s to 4.75s.

2 Background and Related Work

Gawain is based on an epoch-based in-memory database [1], DBX [2]. (In DBX,

© 2015. The authors - Published by Atlantis Press

the snapshot number plays the role of an epoch.) It uses optimistic concurrency control (OCC) [4] to provide serializability and multi-version to support read-only transactions. These conceptions of in-memory databases and the related working about how to support durability for in-memory database will be introduced below.

Multicore In-Memory Database. OCC is a popular concurrency control scheme to provide transaction serializability. Compared with 2-phase locking (2PL), OCC does not block the transaction's execution, thus it can enjoy better scalability when there are few conflicts among concurrent transactions. Gawain is based on an in-memory database with OCC [2] to show that our durability mechanism has little impact on scalable in-memory databases.

In OCC, to avoid contention on the centralized transaction id, recent in-memory databases [2] resort to using version numbers to track the updates of a record. Each record has a version number which will be updated for each modification. Thus, the version number can give the updates' order of a single record. Based on this observation, Gawain can log or recover the system without considering the relationship of concurrent transactions.

Epoch-based in-memory database [1] is a variant of multi-version databases [5]. This mechanism is used to provide consistent snapshot for read only transactions or collect garbage records. Epoch represents a time period and is assigned with an epoch number. The global epoch number is visible to all threads and increased by a single thread periodically. After the advancing of the epoch, each transactional update will create a new version of the record with current epoch number. In OCC, all updates of the same transaction will be committed atomically, thus they will be assigned with the same epoch number. At the same time, if transaction Ta depends on transaction Tb, then Ta's epoch number is not more than Tb's. As a result, each epoch can provide a consistent view of the database.

Durability of Multicore In-Memory For in-memory database, all data residing in the non-volatile memory introduces more challenges to provide the durability.

Aries [6] is a classical database recovery method. Each operation will be logged logically and physically on the disk. To recover the system from a crash event, Aries requires several passes of log to finish the redo and undo. However, it causes large overhead in runtime for high performance in-memory databases. Recording both before and after image leads to large log size, and the throughput is limited by disk I/O. Moreover, the undo process, which demands more than one scan of log in recovery increases the recovery time.

Malviya [7] uses logical log in an extreme way. Only transaction's id and parameters are logged. During recovery, all transactions in log are redone. As log size is minimized, the runtime overhead is little. However, redoing transactions lengthens recovery time, especially when there is much computation in transactions.

SiloR [9] is an epoch-based recovery scheme. Transaction results are logged asynchronously and returned to users in batches, which is similar with Gawain. It also utilizes value logging to facilitate recovery parallelism. However, value

logging has the drawback of logging more data compared with operation logging. Gawain tackles the problem by delta logging which only logs updated columns. SiloR reduces checkpoint sizes by avoiding storing a record in the checkpoint that will be replayed by the log [9], while Gawain does it in another way that only records updated since last checkpoint are recorded in the current checkpoint. Both systems use parallel recovery to reduce recovering time.

3 Design

To reduce the impact on running system, we use delta logging and lightweight checkpoints to minimize the size of log. In case of failure, the system can be recovered from the checkpoints in parallel with good scalability.

Log Algorithm. All the updates of a committed transaction will be logged on the disk before returning to the user. Because the local write set keeps the transaction's updates, after a transaction commits, the records kept in its write set will be flushed to the disk.

One goal of Gawain is to minimize the runtime overhead. In-memory databases with high throughput produce huge amount of updates in a short time. Recovery logging has become one of the major performance bottlenecks due to disk operations. Since more disk writes lead to lower performance, the log size should be reduced to relieve the burden on disk I/O. This is done by using delta logging which only logs the delta of the updates. It means that when a record is updated, only the updated columns are recorded with their column ids. Thus, the log size is reduced, which can benefit the runtime performance.

Worker threads cooperate with logger threads in logging process. Every worker thread keeps a local memory buffer and every logger thread has a log buffer queue. The head of a log buffer stores its content length and epoch. After a worker thread commits a transaction, all the updates in the transaction are converted to log entries and written into its log buffer. If it is the first transaction logged in the buffer, the log buffer's epoch is set to the transaction's epoch. When the buffer is full or the global epoch number is advanced, the local buffer will be passed to the logger thread and appended to the logger thread's buffer queue. Then, a new log buffer will be allocated. The logger thread flushes the buffers in the queue in a while loop. The logging process will keep a durable epoch, Ed. It is updated when all logs of transactions with epoch Ed have been flushed to disk. The durable epoch is written to disk every time it is updated. After that, transactions with epoch Ed can be returned to clients.

Checkpoint. Checkpoint is widely used to reduce the recovery time. On one hand, checkpoint should reflect the recent state of a database. On the other hand, checkpoint should not be taken frequently to avoid hurting the running system. We find that it is not necessary to guarantee the consistency for a single checkpoint and only the record updates after the previous checkpoint need to be reflected on the current checkpoint. Based on these observations, we develop an efficient checkpoint algorithm which releases the consistency and leverages the epoch mechanism to give boundary of a checkpoint.

Gawain keeps a checkpoint epoch, which is initially -1. When a checkpoint

begins, it acquires the checkpoint epoch (Eoldc), and then updates it to Enewc, which is the current checkpoint's own epoch. During recovery, all log records before the newest checkpoint's epoch will be disregarded. Enewc should meet the following requirements: 1. all transactions with an epoch smaller than or equal to Enewc have been committed. 2. Enewc should reflect the most recent state of the database. To calculate Enewc, we leverage the method to calculate the epoch number for consistent snapshot in DBX. It picks the maximum epoch number on which no read-write transactions will commit updates to support consistent snapshot. When the checkpoint is finished, its epoch number (Enewc) will be written to disk.

During the application running, multiple checkpoints will be taken. To avoid redundant information among different checkpoints, only records updated after the previous checkpoint should be reflected in current checkpoint. However, it is challenging to track all the modified records since the last checkpoint. We leverage the checkpoint epoch to identify the record values which already exist in previous checkpoints. During taking the checkpoint, only records with epoch number which is larger than Eoldc will be recorded. For deletion, the deleted record can be garbaged collected only after it exists on some checkpoint.

Gawain guarantees that if the updates reside on the checkpoint, then it should already be recorded in the log and the corresponding transaction is durable. This is done by completing the checkpoint only after the durable epoch Ed is larger than the maximal epoch number a checkpoint observes.

Recovery. In case of failure, Gawain recovers the system from the checkpoints instead of scanning all logs from the beginning. At the same time, multiple recovery threads can process the checkpoints and logs in parallel.

A reader thread and multiple recovery threads cooperate to recover the system in Gawain. The reader threads will gets the newest checkpoint's epoch Ec and the last durable epoch, Ed from disk. Then, it will load the logs and checkpoints from disk into memory. A number of recovery threads receive the log entries from the reader thread and recover the system concurrently.

Since each checkpoint only keeps part of system state, all checkpoints should be loaded by the reader thread. Log entries with an epoch number smaller than or equal to Ec are skipped, as the updates in them are already covered by checkpoints. Log entries with an epoch number larger than Ed are also skipped as recovering the updates in them cannot guarantee the consistency of the recovered system.

The checkpoint entries and log entries are processed similarly in Gawain. It will firstly try to get the record from the recovered database with primary key. If the record does not exist, then it needs to insert the record with the record value or column values in the entry. At the same time, the version number is attached to the corresponding columns. The columns absent in the entry are attached with -1 as their version number. For delete operation, an empty record is inserted with a logical deletion tag and all columns are attached with the version number in the entry. If the record already exists in database, Gawain will compare the version number of the updated columns with the version number in the log entry. If the log entry has larger version number, then the content of the record will be

updated and the version number for the corresponding columns are updated. Otherwise, the log entry is ignored. For example, when Gawain processes a log entry which indicates an update operation on column C1 of record Ra and the log entry has a version number of 10. When it checks the recovering database, it finds that Ra has already existed in the database and C1 is attached with a version number 5 which is smaller than 10. Then Gawain will update C1 with the recorded content and update the version number to 10.

4 Evaluation

All experiments were run on an Intel Haswell i7-4770 processor clocked at 3.4GHz, which has a single chip with 4 cores/8 hardware threads and 32GB RAM. Each core has a private 32 KB L1 cache and a private 256KB L2 cache, and all four cores share an 8MB L3 cache. The machine has two 128GB SATA-based SSD devices.

We use a standard OLTP benchmark: TPCC [8]. By default, 8 million transactions are executed concurrently to evaluate the system performance and 2 logger threads are used. For experiments with checkpoints, all checkpoints distribute evenly during the running process.

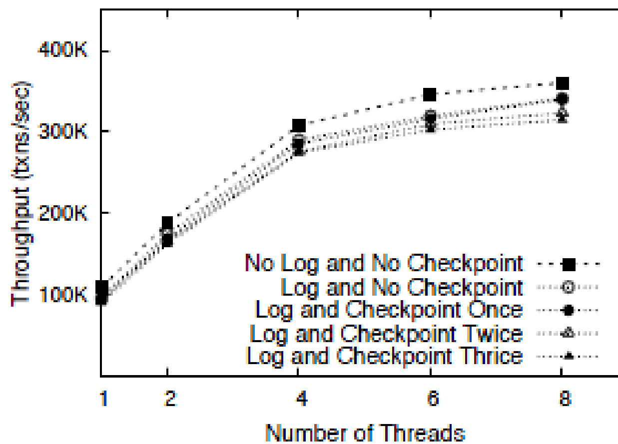

Fig. 1: Throughput of DBX

Runtime performance. The throughput and latency are critical to OLTP applications. Our goal is to have a fast recovery schema while keep the runtime overhead low. Here, we present Gawain's runtime performance.

We experiment with DBX and scale the number of threads from 1 to 8. Throughputs of doing checkpoint 1 to 3 times during the transaction execution are tested and compared with no durability support and no checkpoint support. For different number of threads, the checkpoint size and log size are close, as the number of transactions executed are the same. This setting helps observing the scalability without interfered by the amount of disk writes. Fig. 1 shows that the system scales well under all configurations from 1 to 4 threads. From 4 to 8

threads, the performance scales up slower due to the effect of hyper threads. DBX with no durability achieves a throughput of 361033 transactions/second at 8 threads. Logging causes about 5% slowdown. Injecting one checkpoint brings negligible further slowdown. This is because disk writings are mainly occupied by logging operations. Running two and three checkpoints leads to 4.5% and 2.5% more slowdown. So, even with 3 checkpoints injected, the throughput is not greatly hurt and the system's scalability is kept.

The latency when running 8 worker threads is also tested and the result is shown in Fig. 2. With the number of checkpoints increases from 0 to 3, latency rises gradually due to more disk writes caused by more checkpoints. The latency increases only about 30% by running 3 checkpoints.

Recovery Performance. We use the log and checkpoint files of running 8 working threads to do recovery experiments. To show the performance of parallel recovery, we scale the number threads from 1 to 8.

Fig. 3 shows that the DBX recovery time of using 0 to 3 checkpoints. Compared with no checkpoint, running one checkpoint reduced the recovery time by 37% at 8 threads, since half of the transactions' updates are covered by checkpoint. Adding more checkpoints further shortens recovery time. In all cases, recovery time decreases with the increase of the number of threads. For 1 thread, the log reading and record recovery are done by a single thread. For 2 threads, a recovery worker thread is added, so the reading and recovery are executed in a pipeline way. Since the recovery time is dominated by record recovery, the time saved is not so much. However, from 2 threads to 4 threads, the number of recovery worker thread increases from 1 to 3 and the recovery time decreases dramatically. From 4 to 8 threads, as hyperthread is used, the improvement is not so great, but it still shows some scalability. After all, parallel recovery has a recovery speed 3.7 times of single-thread recovery, with 3 checkpoints.

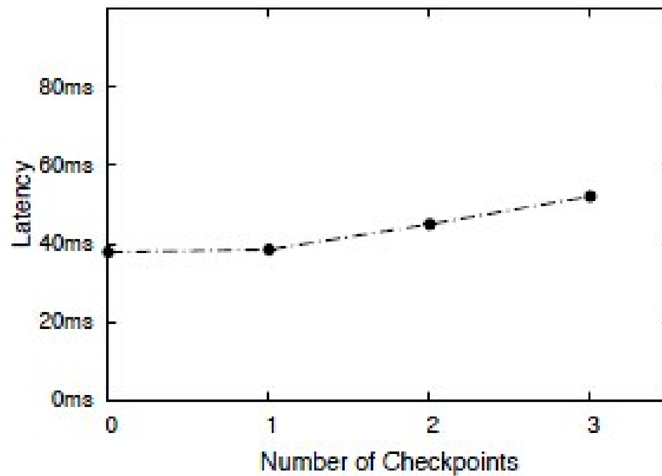

Fig. 2: Latency of DBX.

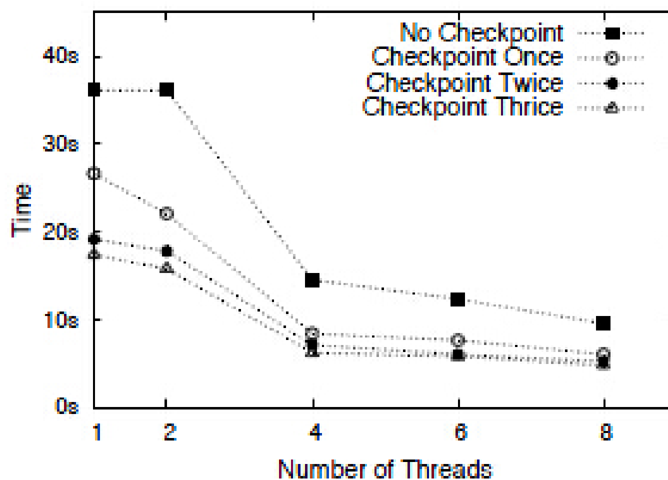

Fig. 3: Recovery Time of DBX.

5 Conclusions

In-memory databases need fast recovery to match its high performance. This paper presents a parallel recovery scheme with high scalability. The records' version numbers are used to decide the before-after relationship of log entries and recover the database back to a consistent state. Also, checkpoint and log size are reduced leading to little overhead on runtime throughput and latency. The recovery scheme is integrated to DBX and shows good performance in recovery.

References

[1] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden: *Speedy transactions in multicore in-memory databases.* The Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013.

[2] Z. Wang, H. Qian, J. Li, and H. Chen: *Using restricted transactional memory to build a scalable in-memory database.* The Ninth European Conference on Computer Systems. ACM, 2014.

[3] V. Leis, A. Kemper, and T. Neumann: *Exploiting hardware transactional memory in main-memory databases*. IEEE 30th International Conference on Data Engineering. IEEE, 2014.

[4] H.T. Kung and J.T. Robinson. ACM Transactions on Database Systems (TODS). ACM, 1981, 6(2):213–226.

[5] P.A. Bernstein and N. Goodman. ACM Transactions on Database Systems (TODS). ACM, 1983, 8(4):465–483.

[6] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ACM Transactions on Database Systems (TODS). ACM,1992,17(1):94–162.

[7] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker: *Rethinking main memory OLTP recovery*. IEEE 30th International Conference on Data Engineering. IEEE, 2014.
[8] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, 2007.
[9] W. Zheng, S. Tu, E. Kohler, and B. Liskov: *Fast databases with fast durability and recovery through multicore parallelism.* In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation. USENIX Association, 2014.