Integration of Parallel Write Ahead Logging and Cicada Concurrency Control Method

Takayuki Tanabe*, Hideyuki Kawashima†, Osamu Tatebe†
*College of Information Science, University of Tsukuba
Email: tanabe@hpcs.cs.tsukuba.ac.jp
†Center for Computational Sciences, University of Tsukuba
Email: {kawasima, tatebe}@cs.tsukuba.ac.jp

***Abstract*—We proposed the idea of combining the Cicada concurrency control mechanism with P-WAL (parallel write-ahead logging) and experimentally evaluated the proposed control mechanism. Our results show that P-WAL can be combined suitably with Cicada without decreasing Cicada's performance. We then experimentally evaluated the most optimal parallel write-ahead logging parameter among (1) conservative lock release; (2) early lock release, the early release of locks on database objects; and (3) group commit, which writes the transaction log to storage all at once. The results show that early lock release and group commit were the most optimal parameters for the combination of Cicada and P-WAL.**

***Index Terms*—big data, transaction, concurrency control, Cicada, write-ahead logging, logging, P-WAL**

I. Introduction

A. Background

Transaction processing [1] [2] is used everywhere. It is necessary for credit card payment, which is the foundation of economic activity, and it is issued in the depths of computer operating systems when we save a data. The transactions processed by humanity are increasing after year, which requires that they are processed efficiently.

Transaction processing systems are composed of concurrency control mechanisms [3] and logging mechanisms [4]. Concurrency control mechanism controls the concurrently accesses of multiple transactions to the database and guarantees the isolation of ACID which has to be satisfied by transactions. Logging mechanism writes log records of transactions to persistent storage before an update by the transaction is reflected in the database.

Recently-proposed concurrency control mechanisms include FOEDUS [5] and TicToc [6], as well as Cicada [7]. Cicada is recognized as one of the state-of-the-art concurrency control mechanisms. It combines multi-version [8] [9] [10], optimistic [11] [12], and multi-clock [6]. Cicada's design can reduce the overhead and contention of transaction processing. Cicada's optimistic multi-version reduces both memory-access-level interference and transaction-level conflicts between concurrent transactions. Multi-clock does not have a conventional bottleneck by centralized timestamp allocation.

Recently-proposed write-ahead logging mechanisms include Aether [13], passive group commit [14], SiloR [15], and P-WAL [16]. Also, there are many researches assuming flash devices and NVRAM [5] [14] [17] [18] [19] [20]. A key feature of these techniques is to exploit the parallelism of multi-core architectures. Aether allows threads to simultaneously copy log records to a centrally shared log buffer on memory. Passive group commit and P-WAL have the log buffer for each thread, and they allow for a thread to write each log buffer to storage devices in parallel. To improve parallelism, these mechanisms use the early lock release (ELR), which releases locks on database objects at an early stage to reduce locking time, and group commit [21] which batches multiple transaction logs together and synchronizes them to persistent storage. Using these two techniques reduces both the contentions with locks on database objects and the I/O cost for persisting the transaction logs.

B. Problem

The paper on Cicada [7] describes the concurrency control mechanism, while the description of logging mechanism is limited to concept. It explains a direction to persist the log records, but it does not discuss the design and the implementation. Therefore, the appropriate logging mechanism in Cicada should be clarified. We said the transaction processing consists of concurrency control and log persistence mechanism. Therefore, to make Cicada a transactional system, it is necessary to examine the optimal log persistence mechanism for Cicada.

C. Proposal

As mentioned above, recent log persistence methods use two techniques: early lock release and group commit. We propose following four combinations of log persistence mechanisms with Cicada.

- Conservative lock release + Non-group commit
- Conservative lock release + Group commit
- Early lock release + Non-Group commit
- Early lock release + Group commit

Table I show four combinations.

P-WAL [16] is a log persistence mechanism that writes the transaction log to storage in parallel. Conservative lock release (CLR) abandons locks on database objects after persisting the log; early lock release abandons them before persisting the log. We experimentally evaluated the proposed methods, and the results showed that P-WAL does not decrease Cicada's performance in a multi-core environment and that the optimal

978-1-5386-4705-9/18/$31.00 ©2018 IEEE
DOI 10.1109/SMARTCOMP.2018.00018

TABLE I
COMBINATION OF WRITE AHEAD LOGGING PARAMETERS

	CLR	ELR
Group commit	CLR + Group	ELR + Group
Non group commit	CLR + Non group	ELR + Non group

parameters for P-WAL are early lock release and group commit.

D. Organization

The rest of this paper is organized as follows. In Section II, we explain the Cicada concurrency control mechanism, write-ahead logging, and parameters for write-ahead logging. In Section III, we propose our method, which combines Cicada and P-WAL. In Section IV, we evaluate the proposed method, and in Section V, we conclude this paper.

II. PREPARATION

A. The Cicada concurrency control mechanism

Cicada is a concurrency control mechanism that combines multi-version, optimistic, and multi-clock. Multi-version makes it possible to avoid contention between read and write locks by creating a new version whenever data is updated. So, it can reduce transaction-level conflicts. Optimistic describes a method of control that emphasizes efficiency on the expectation that transactions will not compete, and it consists of three phases: read, validation, and write. The protocol executes an operation in the read phase without holding a lock, maintains consistency in the validation phase, and it reflects the update in the database in the write phase. Because of unholding a lock, read/write operations execute in not shared memory but thread local memory. So it can reduce memory-access-level conflicts. Multi-clock is a technique for generating timestamps allocated to versions in multiple worker threads without using a centralized clock so that it can reduce conflicts on sequence number acquisitions.

1) Protocol: Figure 1 shows the workflow of our re-implemented Cicada.

As Cicada uses optimistic, its protocol is divided into three phases: read, validation, and write. When a transaction begins, it moves into the read phase, a worker thread calculates a timestamp to allocate to the transaction which is processed by the thread. It is a characteristic of multi-clock that worker threads issue timestamp themselves. It then searches for the target version of the operation, based on the allocated timestamp and executes the operation. Due to multi-version, it is necessary to search for the optimal version in order among multiple versions. At this time, when executing the write, it duplicates that version in thread internal, executes the write there, and keeps the duplicate version in a write-set. It keeps the version targeted for reading in a read-set. It then moves into the validation phase, in which it installs the pending version contained in the write-set globally. At this time, if consistency will not be maintained, it aborts the transaction.

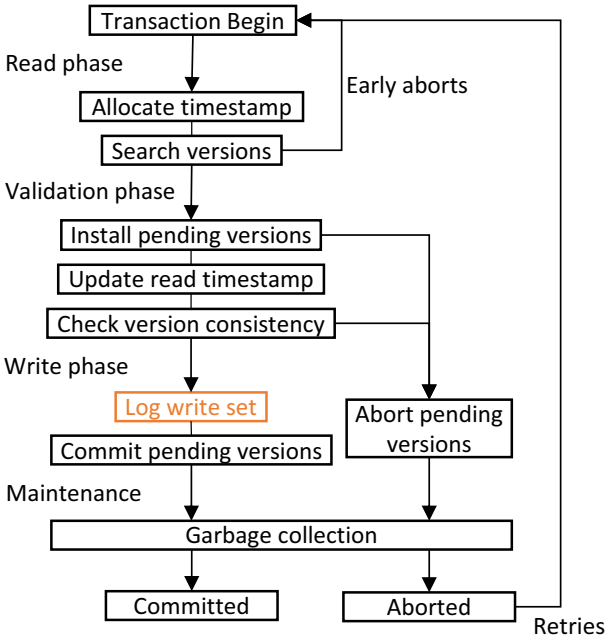

Fig. 1. Workflow of re-implemented Cicada [7]

When the pending version is fully installed, it updates the read timestamp of the version in the read-set. As the version records the maximum timestamp that was read, it will be updated as necessary. When the update is finished, it verifies if the result of the operation is consistent with other workers based on the read-set and write-set. If it is, it reflects the update in the database in the write phase; if not, it aborts the transaction and retries from the read phase.

B. P-WAL (Parallel write-ahead logging)

P-WAL is a write-ahead logging (WAL) protocol suitable when using flash storage as a storage device. Figure 2 shows the architecture of P-WAL.

Unlike conventional HDD-type storage, flash storage exhibits fast parallel random I/O access performance. This is why for P-WAL, we have proposed the WAL protocol, which takes advantage of the concurrent random access performance of flash storage and operates in parallel. The major differences between P-WAL and conventional WAL are the dividing of the WAL buffer and the dividing of the WAL file.

1) Log sequence: In P-WAL, each worker thread has a WAL buffer, which could make the log sequence unclear. To resolve this, it assigns a log sequence number (LSN) to each log record. This LSN is incremented each time when it is issued, which makes it possible to provides a serial feature to the sequence of log records and detect missing log records caused by errors or failures.

2) Notification Scheduling: Even if the protocol has finished persisting the log record, it is not always possible to

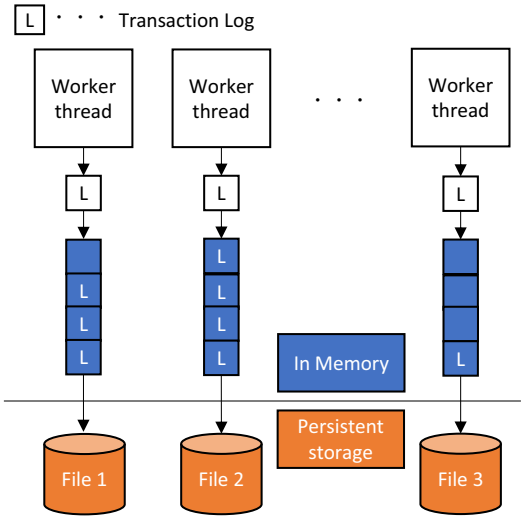

Fig. 2. P-WAL architecture

notify the commit to an application. This is because when the processing of a transaction A depends on the result of another transaction B, the commit condition is not met until the processing of transaction B ends successfully. To control the commit notification, each WAL buffer retains the LSN of the last persisted log record (flushedLSN). The commit condition of a transaction is met when the lowest flushedLSN, among those retained in the WAL files of all worker threads, is greater than or equal to the LSN of that transaction. The worker threads keep transactions that are waiting for commits of other transactions in a queue and checks the flushedLSN of other WAL files whenever a transaction ends. Commit notifications for transactions that satisfy the commit condition are then sent to applications and the transaction is removed from the queue.

C. Parameters for write-ahead logging

1) Group commit: Storage IO was previously a performance bottleneck for transaction processing. In addition, HDD-type storage had characteristically fast sequential access performance but inefficient random access performance. Group commit [21] was proposed to reduce this IO cost and take advantage of the fast sequential access performance. To minimize the number of IO commands sent, worker threads send the logs to logger threads and request logging. The logger threads write the sent logs to storage in a single write operation. Then the transactions processed by each worker thread wait for the log write to complete. This is the method by which group commit improves throughput.

When either of the following two conditions listed below is met, the logger threads write the sent logs to storage in a single write operation and upon completion, notify the thread that issued the transaction.

- The amount of the logs written exceeds a certain threshold
- A certain amount of time has elapsed since the previous write

2) Lock release mechanisms: When a transaction executes an operation to record, it needs to acquire a lock on it before doing so. This lock is released at a different time, depending on the lock release mechanism, of which there are two main types: conservative lock release (CLR) and early lock release (ELR) [13].

- **Conservative lock release:** Releases the lock after persisting the log record.
- **Early lock release:** Releases the lock before persisting the log record.

In ELR, the lock is released before persisting the log record, which shortens the locking time and allows transactions that were waiting for a lock to resume processing early. This should reduce contention with other worker threads for database objects and improve concurrency.

ELR does not guarantee that the database object, executed reads and writes, is persisted. Therefore, the commit condition of the transaction requires that all dependent transactions are committed and that recovery be possible if an error occurs. This means that it waits to commit until all dependent transactions are committed, which makes it necessary to notify other worker threads of the commit.

III. Proposed Method: Combining Cicada and P-WAL

Transaction processing consists of concurrency control and write-ahead logging. The original paper on Cicada [7] states that it supports parallel logging but does not discuss its design and implementation in detail. The performance evaluation was also done with logging disabled, so it is necessary to examine appropriate methods for persisting logs. In this paper, we propose a method of applying persistence processing to Cicada by combining Cicada and P-WAL.

A. Persistence protocol

We propose a protocol to apply persistence processing to Cicada. As in P-WAL, each worker thread has a WAL buffer. As Figure 1 shows, transaction processing is largely divided into read, validation, and write phases. When the validation phase is complete, the worker threads write the transaction log to their own WAL buffers. After that, it persists those transaction logs (written to the WAL buffers) in WAL files, and when persisting is complete, it reflects the changes in the database and releases the locks on the database objects.

P-WAL uses a centralized shared counter to issue an LSN to each log record, but our protocol does not. Instead, it uses the timestamp assigned to each transaction. Cicada's timestamps increase monotonically and have unique features, and this makes it possible to mediate transaction log order.

B. Lock release methods and group commit

1) Lock release mechanisms: Lock release mechanisms include conservative lock release (CLR) and early lock release (ELR) [13].

CLR releases locks on database objects after persisting the log record. Cicada does not read locked tuples during the read phase, and thus, it can guarantee that the values of the tuples that the worker threads read during the read phase are persisted.

ELR releases locks on database objects before persisting the log record. This makes it possible to shorten the lock time as much as the latency of persisting the log. Releasing the lock early reduces contention with other worker threads and contributes to improved performance.

ELR does not guarantee that the values of database objects that the worker threads read during the read phase are persisted. Therefore, the commit condition of the transaction requires that all dependent database objects be persisted. This means that it waits to commit until all dependent database objects are persisted, which makes it necessary to control notification of the commit.

To solve this problem, P-WAL orders the log records by using a shared counter to issue LSN. Each worker thread retains the last persisted LSN (flushedLSN). The transaction looks at the lowest flushedLSN among all worker threads and checks whether or not it is higher than the LSN assigned to the log record. If it is, that guarantees that all older transactions have been persisted, and a commit notification can be sent.

Our protocol uses "precommitted" as the status of versions in tuples. It takes pending versions that are retained before persisting the log, makes their status "precommitted", and the begins persisting the log. When that is complete, it looks at the status of the versions that the retained pending versions are dependent on, and if they are all "committed", then it sends the commit notification. If not, it enqueues the transaction in a transaction queue retained by the worker threads. When the next transaction is complete, it checks the transaction queue, dequeues transactions whose retained pending versions are all committable, and sends commit notifications. When it comes to an uncommittable transaction, it finishes checking the transaction queue.

2) Group commit: Group commit is a mechanism for persisting multiple transaction logs in a single batch, and it can be applied to CLR and ELR.

When group commit is applied to CLR, the number of locks and the locking time increase, since locks are released after persisting the log. Without group commit, deadlocks will not occur, if a lock can be held on the entire sequence, but they can easily occur with group commit, as it spans multiple transactions to hold a lock. The conditions to execute a group commit are as follows:

- The amount of the logs written exceeds a certain threshold
- A certain amount of time has elapsed since the previous write

When group commit is applied and a deadlock state occurs, the execution of a group commit depends on the latter condition.

There are two times when the group commit conditions are checked. The first is upon aborting. In Cicada, after a worker thread waits for a lock for a certain amount of time, it aborts the transactions that it handles. Before executing this abort, it thinks of the possibility that it and a worker thread were waiting for a lock on each other. By aborting after a certain amount of time has elapsed, it can avoid a circular wait, and if group commits can be executed, circular waits can be resolved because locks are released after the log is persisted.

The second time is upon committing. At this time, the two group commit conditions are checked, and if either condition is met, then the group commit is executed.

C. Proposed method: combinations of parameters

Applying (or not applying) CLR, ELR, and group commit to the proposed method gives us the following four methods:

- CLR + Non-Group commit
- CLR + Group commit
- ELR + Non-Group commit
- ELR + Group commit

"CLR + Non-Group commit" uses CLR, but not group commit, so it persists the log for each transaction and then releases the lock on its database object.

"CLR + Group commit" uses CLR and group commit, so it persists multiple transaction logs in a single batch and then releases the locks in a single batch. The group commit is executed when a certain number of transactions have accumulated, or a certain amount of time has elapsed since the last group commit.

"ELR + Non-Group commit" uses ELR but not group commit, so it persists the log for each transaction but releases the lock on the database object before doing so. It controls notifications by checking whether or not all the versions looked up by pending versions contained in the write-set retained in the transaction are committed.

"ELR + Group commit" uses ELR and group commit, so it persists multiple transaction logs in a single batch but release the locks on the database object before doing so. It controls notifications by checking whether or not all the versions looked up by pending versions, contained in the write-set retained in the transaction, are committed.

For methods that use group commit, the earliest timestamp in the transaction log stored in the WAL buffer is recorded. That timestamp is then recorded in each worker thread, and versions older than that timestamp are garbage-collected, except for the newest committed version. For methods that do not use group commit, the timestamps retained in each worker thread are recorded. That timestamp is then recorded in each worker thread, and versions older than that timestamp are garbage-collected, except for the newest committed version.

IV. EVALUATION

We experiments our proposed method in a multi-core environment. We conducted experiments to evaluate the four

TABLE II
TESTING ENVIRONMENT

Processor	Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Number of cores	24
Memory	66GB
OS	CentOS release 6.9 (Final)

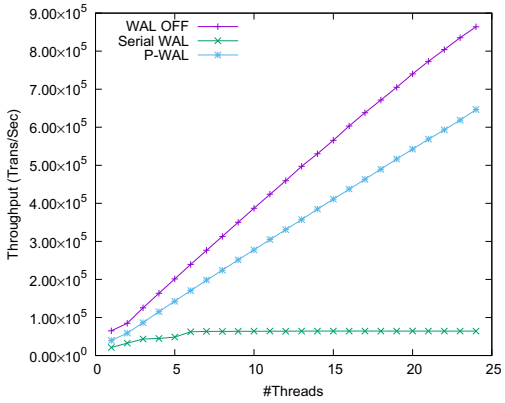

Fig. 3. Evaluation of Cicada + Serial WAL performance

combinations of CLR, ELR, and group commit that we proposed, as well as ELR + group commit + serial WAL. The tests involved each worker thread performing 3 million transactions. We conducted the tests 5 times each and used the averages as plot dots. When the WAL file was written, inserting a fixed latency functioned as a substitute for the write. The experimental environment is shown in Table II.

A. Evaluation of Serial WAL

The performance of ELR + group commit + serial WAL was compared with that of not logging. The workload was set at 20 write operations for one transaction, the fixed latency set at 50 ms. The test results are shown in Fig. 3.

The results showed that combining Cicada with serial WAL decreased performance. In WAL, all worker threads share one WAL buffer, so we think that contention for the WAL buffer for this. The results also showed that Cicada's performance scales up when logging is disabled.

B. Evaluation of write-ahead logging parameters combined with P-WAL

We compared our four proposed methods with a workload of 1 million tuples and a storage IO of 50 us. Group commit conditions were defined as (1) when 5 transactions have accumulated in the transaction log, or (2) when 500 us had elapsed since the last write. Figs. 4 and 5 show the results of these tests.

Fig. 4 shows the performance with only write-operation transactions; Fig. 5 shows the performance when 80% of the transactions were read operations.

The results show that early lock release + group commit is the optimal method. We describe our observations regarding

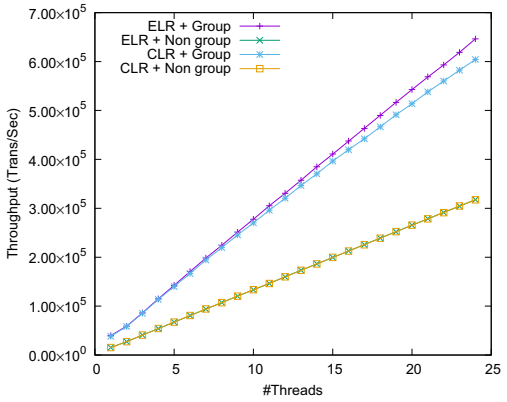

Fig. 4. Write only transaction, The number of tuples is one million.

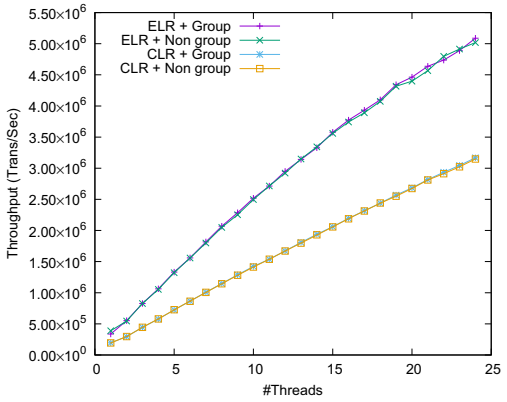

Fig. 5. Write operation rate is 20%, Read operation rate is 80%, The number of tuples is one million.

group commit. The use or non-use of group commit effects a significant in performance. If group commit is not used, the log is persisted with each transaction, and as storage IO is set at 50 us, it will take at least 50 us for one thread to process one transaction. On the other hand, using group commit reduces the storage IO cost in proportion to the number of transactions batched in a single commit. For example, storage IO can be reduced to as little as 10 ms per transaction when this number is set to 5, as the log is persisted after 5 transactions are processed. This means that using group commit results in better performance.

Here is the reason why ELR (early lock release) exhibits superior performance. By releasing the lock before persisting the log, ELR allows transactions that were waiting for that lock to proceed. However, CLR (conservative lock release) releases the lock after persisting the log and thus continues to hold the lock for as long as it takes to persist the log. As a result, transactions waiting for this lock have to wait for the completion of transactions retaining a lock in order to be persisted. That is why we believe CLR and ELR show these results.

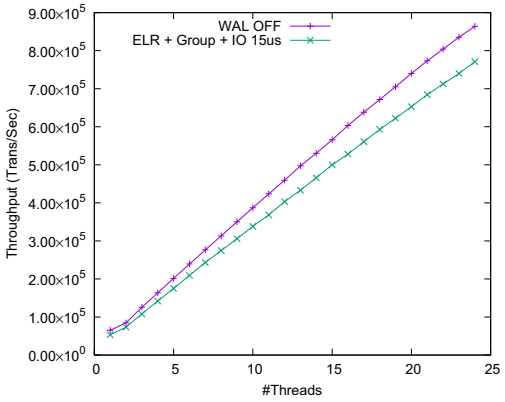

Fig. 6. Assuming NVRAM as storage IO

1) Evaluation assuming NVRAM: The results so far suggested that P-WAL + early lock release + group commit is the most optimal method. We then used this method to experiment with assuming NVRAM as the persistent storage. We assumed NVRAM by setting the storage IO at 15 us. The workload was 1 million tuples of write-operation-only transactions. The performance was compared with Cicada with logging disabled, and the results are shown in Fig. 6.

The results show that when NVRAM was assumed to be the log write destination, enabling logging does not decrease Cicada's performance.

V. Conclusion

In this paper, we examined the Cicada concurrency control mechanism. The original paper on Cicada states that it shows high performance under various workloads by combining multi-version, optimistic, and multi-clock, reducing overhead and contention of transaction processing.

We were interested in Cicada for one reason: it is the optimal mechanism for log persistence. Transaction processing systems consist of concurrency control mechanisms and logging mechanisms. The original paper on Cicada [7] states that it supports parallel logging but does not discuss its design and implementation in detail, leaving it unclear as to what the most appropriate method might be. To that end, we combined Cicada with the parallel write-ahead logging mechanism P-WAL [16] and experimentally evaluated it.

The results of our experiments showed that (1) serial WAL decreases Cicada's performance, (2) early lock release and group commit are the optimal write-ahead logging parameters when combining Cicada and P-WAL, and (3) even in an environment that assumes the non-volatile memory as persistent storage, this optimal method does not decrease Cicada's performance.

In this paper, we conclude that, when combining Cicada and P-WAL, early lock release and group commit do not decrease Cicada's performance in a multi-core environment and when using non-volatile memory in the same environment.

Acknowledgement

This work is supported in part by JST CREST JPMJCR1414, JST CREST JPMJCR1303, KAKENHI Grant-in-Aid No. 16K00150, and KAKENHI Grant-in-Aid No. JP17H01748.

References

[1] J. Gray, A. Reuter, and M. Kitsuregawa, *Transaction processing: concepts and methods*. Nikkei BP Sha, 2001.

[2] G. Weikum and G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems," 1987.

[4] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, no. 1, pp. 94–162, 1992.

[5] H. Kimura, "Foedus: Oltp engine for a thousand cores and nvram," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 691–706.

[6] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1629–1642.

[7] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 21–35.

[8] D. P. Reed, "Implementing atomic actions on decentralized data," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 1, pp. 3–23, 1983.

[9] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1243–1254.

[10] K. Kim, T. Wang, R. Johnson, and I. Pandis, "Ermia: Fast memory-optimized database system for heterogeneous workloads," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1675–1687.

[11] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[12] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 49–60, 2016.

[13] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: a scalable approach to logging," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 681–692, 2010.

[14] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[15] W. Zheng, S. Tu, E. Kohler, and B. Liskov, "Fast databases with fast durability and recovery through multicore parallelism." in *OSDI*, 2014, pp. 465–477.

[16] K. Kamiya, H. Kawashima, T. Hoshino, and O. Tatebe, "Parallel write-ahead logging method p-wal," *Database of journals of the Information Processing Society of Japan (TOD)*, vol. 10, no. 1, pp. 24–39, 2017.

[17] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.

[18] S. Chen, "Flashlogging: exploiting flash devices for synchronous logging performance," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 73–86.

[19] S.-W. Lee and B. Moon, "Design of flash-based dbms: an in-page logging approach," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 55–66.

[20] ——, "Transactional in-page logging for multiversion read consistency and recovery," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 876–887.

[21] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, "Group commit timers and high volume transaction systems," *High Performance Transaction Systems*, pp. 301–329, 1989.