# Response to Reviews for "*Improving OCC Performance Through Transaction Batching and Operation Reordering*"

Bailu Ding, Lucja Kot, Johannes Gehrke

We would like to thank the reviewers and the meta-reviewer for their insightful comments and suggestions. Below is a summary of the changes we made in response to these comments. We have fixed typos and omit such comments in the response. Changes are marked in blue in both the response and the revised submission.

## 1 Changes based on meta-reviewer comments

We first summarize and address the major comments from the reviews.

(R1) *Expand the experimental evaluation to include a comparison with other high-performance OLTP engines that use OCC.*

We integrated our techniques to Cicada and compared its performance with Cicada, Silo, TicToc, and ERMIA. Our techniques achieved at least 2x throughput on write-intensive YCSB benchmark compared with these state-of-the-art OLTP kernels.
The experiment details are described in Section 6.7.

(R2) *Fix typos, unclear parts, and expand discussion per the reviewers requests.*

We fixed all the typos mentioned by the reviewers. (TODO)

## 2 Changes based on individual reviewer comments

In this section we explain how we addressed individual reviewer comments which are not directly subsumed by a metareviewer comment. We have addressed comments on clarifications of the writing, and omitted similar comments that can be addressed by previous responses.

### 2.1 Reviewer 1

(R1.1) *Add an experiment to show the performance of the proposed techniques in systems where the clients do not send their next transaction until their previous one is finished.*

We integrated our techniques to Cicada, where each thread processes a transaction synchronously. Please see R1 for details.

(R1.2) *DBMS-X Add a more detailed description regarding the DBMS-X experiment.*

Our prototype issues transactions to DBMS-X via JDBC, where it can submit transaction statements individually or in batch. Either way, the connection will block until all the transactions get processed, i.e., commit or abort. As DBMS-X receives a batch of transaction statements from a connection, it can execute them concurrently. Batching transaction statements reduces the overhead of communication and can increase the concurrent level of DBMS-X as it receives more active transactions.
We added a more detailed description of why batching helps DBMS-X in Section 6.7.

(R1.3) *Improve the legends of the figures.*

We updated the legend to replace unnecessary abbreviations, e.g., bch -¿ batch.

(R1.4) *Add an experiment to demonstrate to generality of the sort-based algorithm by using more policies or benchmarks.*

We agree that the policies should be customized based on the objective of transaction processing system and its architecture.

We further extended our sort-based framework to incorporate a thread-aware policy for an important class of multi-threaded OLTP architecture, where each transaction is processed independently and synchronously by a dedicated thread.

We implemented the policy on top of Cicada and please see Section 6.7 for the evaluation details.

## 2.2 Reviewer 2

(R2.1) *Add a discussion which explains what constraints on the system make the system best suited for integration with an in-memory versioned key-value store and what would need to change to be integrated with other stores.*

We retreated this statement from the introduction. In our experiments, we integrated our techniques to two systems that are not based on key-value stores, i.e., Cicada (supporting hash index and B+ tree index) and DBMS-X (using BW-tree). The result shows that a variety of storage and transaction processing architectures can benefit from our techniques.

(R2.2) *There are many moving parts in the experiments so it is difficult to picture the tradeoffs holistically. I suggest that either the results are represented in a formula or represented in a table showing the tradeoffs.*

**[TODO!: Add to appendix]** We added a table to summarize the trade-off of different parameters and policies in the Appendix.

## 2.3 Reviewer 3

(R3.1) *The papers contains a bag of ideas/optimizations, arguably unrelated, based on known techniques, so the overall contributions and novelty is limited.*

**[TODO!: Better wording needed]**

- Our contribution is a general framework of batching and reordering for OCC based OLTP systems.
- The framework uses a sort-based policy to customize for different transaction processing performance objectives and system architectures.

(R3.2) *Although the evaluation is comprehensive and detailed, but the authors only presented a micro benchmark, a self-comparison without considering other state-of-the-art approaches. Here are few important related CCs (related work discussion can also take into these approaches as well)*

Please see R1 for details.

(R3.3) *The authors provide a black box comparison of their approach in DB-X, but the basis of the comparison is unclear (also not sure what does "Good Throughput/Transaction" mean). Although the effort is appreciated, it would be much better to compare with relevant, disclosed existing algorithm, or at the very least, the concurrency of the model of DB-X is carefully explained. A black box graph adds no value.*

"Good Throughput" means the number of committed transactions per second, since some literature refers "Throughput" as the number of transactions processed per second, regardless either they commit or abort. We changed to term to "throughput" and clarified its semantics here in Section 6.1.

We added an experiment to compare with other state-of-the-art OLTP kernels. Please see R1 for details.

DBMS-X is Hekaton in Microsoft SQL Server. Due to confidentiality, we could not disclose the name of the system or too much details of the system to make it identifiable.

(R3.4) *It is unclear why the authors choose to have 300 active transactions, which would imply the need for having 300 physical threads. I further suppose, the authors considering in-memory implementation given all OLTP DB can fit entirely in memory today. If in fact, the number of active transactions is larger than threads, then there will be many context switches, and frankly, the whole setting would be questionable, which could also explain why the overall throughput is never exceeded half-million transactions/second.*

As described in Section 6.1, our prototype for micro benchmark experiment is implemented with asynchronous transaction processing architecture. Each thread multiplexes multiple transactions simultaneously to mask the latency of transaction workflow execution, IO accesses, and network communication. Thus, the number of threads can be much less than the number of transactions. This architecture has loosely coupled components and can scale storage and compute independently, which is especially applicable for elastic transaction processing in the cloud.

We also integrated our techniques to Cicada, where each thread processes a transaction synchronously and independently. In our integration with Cicada, each thread is pinned to a dedicated physical CPU core and the number of active transactions never exceeds the number of cores. We achieved close to 2 millions transactions per second with highly skewed, write-intensive workload, which is at least 2x the throughput of the state-of-the-art OLTP kernels.

(R3.5) *In some graphs, x-axis label is cut off.*

We fixed the truncated labels and ticks.