On Main-Memory Multicore Transaction Performance

Yihe Huang
yihehuang@g.harvard.edu
Harvard University
Cambridge, MA

1 Introduction

Main-memory database systems achieve high throughput by keeping their data structures entirely in memory, and by taking advantage of parallel execution on multicore hardware. This requires concurrency control (CC), which ensures serializability (or other isolation properties) for concurrently executed transactions. The performance of CC mechanisms has been an active area of systems research. Many different CC mechanisms claim to improve performance over baselines such as optimistic concurrency control (OCC).

However, main-memory transactions can run so fast that small differences in system design and implementation, often unrelated to CC mechanism, can have magnified impact on performance. For instance, many CC schemes have been motivated by apparent collapse under high contention of the performance of single-version OCC; but when using our own optimistic OCC-based system, we do not observe this collapse. Why is this? And are the observed benefits of new CC designs, such as multi-version and mixed optimistic/pessimistic CC variants, due more to CC mechanisms, or more to implementation differences?

In this work, we identify non-CC implementation factors that can have nontrivial impact on transaction performance, and conduct a set of carefully controlled experiments to find their degree of impact. We find that for a high contention OLTP workload, the collapse of other OCC implementations' performance is caused by non-CC factors such as index contention, index type, and contention regulation. In particular, the index contention we found is present in many systems and has severe adverse performance implications, but is largely overlooked by recent work.

2 Background and Related Work

Concurrency control (CC) is an old topic in database research, and it has seen renewed interest recently due to high transaction throughput achieved by modern main memory database systems. As with many other database properties, there is no single "best" CC algorithm, and the choice of a best CC can depend on the workload.

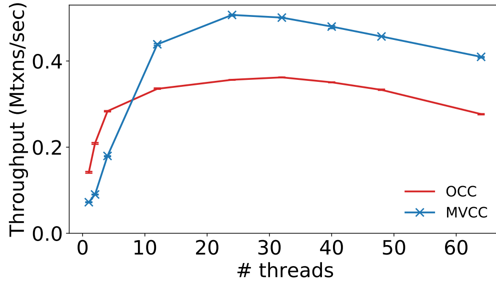
One popular and well-studied CC is optimistic concurrency control (OCC) [5]. It is used in many modern main memory database systems [2, 4, 12] due to its simplicity. It is also well understood that OCC performs well when transactions are expected to rarely conflict, due to OCC eliding writes to shared memory for transactional reads. OCC's limitations are clearest in workloads with high contention, where repeated conflicts and rollbacks can cause starvation or livelock.

Much modern CC research aims at addressing these limitations. For example, mixed OCC/locking protocols such as MOCC [13] and ACC [11] aspire to take advantage of pessimistic CC's progress guarantee at high contention and OCC's low overhead at low contention, thereby achieving the best of both worlds. These adaptive algorithms use heuristics to estimate the level of conflicts in the workload and dynamically switch between pessimistic and optimistic modes of operation. Multi-version concurrency control (MVCC) [10], another popular technique, stores multiple copies of the record to greatly reduce read–write conflicts between transactions. Modern MVCC systems like Cicada [6] claim to match the performance of their OCC counterparts while keeping MVCC's advantage in handling long-running read-only queries. Novel OCC-based systems like TicToc [14] apply MVCC-style timestamp ordering in a single-version setting, thereby reducing the overhead of serializability validation without paying the overhead cost of multiple versions. All of these systems claim to outperform existing OCC systems on realistically high-contention workloads, often by significant margins.

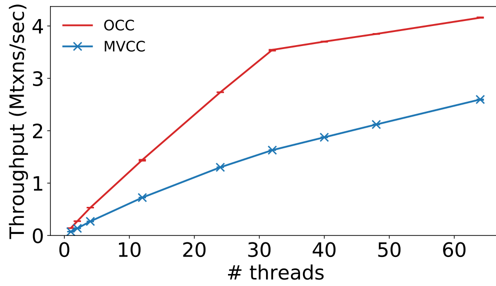
3 Results

We conducted our experiments on STO [3], a software transactional memory, using OCC and MVCC variants. The experiments are run on Amazon EC2 m4.16xlarge dedicated instances powered by two Intel Xeon E5-2686 v4 CPUs (16c/32t each, 32c/64t per machine) with 256GB of RAM. We implemented a simple main-memory database using ordered and unordered index structures in STO, and implemented a TPC-C benchmark on top of this main-memory database. We expected to replicate results reported by Cicada [6], which evaluates several CC schemes at different contention levels. Cicada reports that OCC performance on high-contention TPC-C declines by roughly 50% as the number of threads rises from 1 to 28, and that OCC and MVCC perform similarly on low-contention TPC-C.

Our results, shown in Figure 1, contradicted this expectation. High-contention OCC performance *rises* by roughly



(a) High contention (1 warehouse)



(b) Low contention (# warehouses = # threads)

Figure 1. TPC-C full mix for OCC and MVCC.

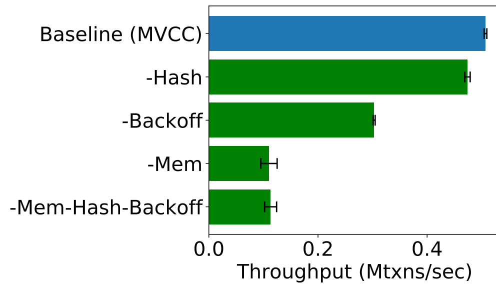


Figure 2. Factor analysis in TPC-C full mix; MVCC STO, 1 warehouse, 24 threads. All results shown have the index contention fix applied, without which the performance would completely collapse.

50% from 1 to 28 threads, declining modestly thereafter; and under low contention, OCC significantly outperforms MVCC (as well as Cicada).

As we investigated further, we found that these results were largely explained by differences in implementation factors other than core CC. These factors had dramatic impact on the apparent performance of OCC, and differences in the way these factors are implemented can all-too-easily be mistaken for fundamental differences in performance of the underlying CC mechanisms.

The most important factors we observed include:

Index contention: When an index uses a multi-part key (such as a pair of country and region) where range operations are common (such as scans over a country), it's important to design the index to reduce conflicts. In many implementations, a multi-part key is simply marshalled into one big key in some order-preserving but otherwise opaque way. In such index implementations range scans can easily conflict with operations such as inserts on distinct ranges. These false conflicts contribute significantly to performance collapse when phantom protection is enabled. False conflicts can be discouraged by mapping different key parts to distinct index sub-structures. Our (ordered) indexes are implemented using Masstree [7], which supports this key partitioning very easily.

Contention regulation refers to the reaction taken by a thread after a transaction aborts. Over-eager transaction retries can damage performance on multicore machines by causing repeated cache invalidations. As with spinlock implementations and even network contention [1, 8], exponential backoff after aborting provides good balance between fast retries after rare contention events and low overhead during high contention. Furthermore, OCC, which aborts frequently, should implement aborts in an efficient way. Some systems, such as Silo [12], used expensive programming language constructs for aborts—in fact, aborts induced lock contention on an underlying language runtime object! Our baselines use exponential backoff with efficient aborts.

Index types refers to a system's choice of data structure for indexes: it's more efficient to use hash tables for indexes that do not require range queries.

We also observed that for MVCC, **memory allocation** is another important factor, because every update requires creating a copy of the record. A good multicore memory allocator should at minimum take advantage of superpages and impose little contention of its own. We use a fast general-purpose memory allocator, rpmalloc [9].

Figure 2 shows the degree of impact of non-CC factors discussed above in a high contention TPC-C workload. We show results for MVCC only but they are also representative of OCC (except for the allocator).

4 Conclusion

In our work, we showed that the the potential for performance collapse of OCC may have been exaggerated in prior work. Non-CC implementation factors actually contributed to this perceived collapse, and these factors can have just as much or more impact on performance than CC. We listed our implementation choices for reference.

Our results also show the importance of controlling for non-CC factors when comparing different CC implementations. Care must be taken when drawing conclusions from cross-system comparisons or comparisons with a system re-implemented from its text description.

References

[1] Norman Abramson. 1970. THE ALOHA SYSTEM: Another Alternative for Computer Communications. In *Proceedings of the November 17-19, 1970, Fall Joint Computer Conference (AFIPS '70 (Fall))*. ACM, New York, NY, USA, 281–285. https://doi.org/10.1145/1478462.1478502

[2] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. ACM, 54–70.

[3] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*. ACM.

[4] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 691–706. https://doi.org/10.1145/2723372.2746480

[5] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[6] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 21–35. https://doi.org/10.1145/3035918.3064015

[7] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, 183–196.

[8] John M Mellor-Crummey and Michael L Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9, 1 (1991), 21–65.

[9] Rampant Pixels. 2019. rpmalloc - Rampant Pixels Memory Allocator. https://github.com/rampantpixels/rpmalloc

[10] David Patrick Reed. 1978. *Naming and synchronization in a decentralized computer system.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[11] Dixin Tang, Hao Jiang, and Aaron J Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All.. In *The 8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*.

[12] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, 18–32.

[13] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment* 10, 2 (2016), 49–60.

[14] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 1629–1642.