Cascade: Optimal Transaction Scheduling for High-Contention Workloads

Tim Baccaert (September 2025)
Supervised by Prof. Dr. Bas Ketsman
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
tim.baccaert@vub.be

***Abstract*—The performance of multi-core transactional systems is a well-studied area, with the ultimate goal of optimizing the balance between high concurrency and the appearance of serial execution. In recent years, partitioning-based systems have shown that we can gain benefits from ahead-of-execution analysis on batches of transactions. However, this benefit is largely lost as contention increases.**

In this work, we investigate what it means for a batch to be executed optimally in the face of high contention. Intuitively, we will consider a schedule to be optimal if it minimizes the time spent waiting for exclusive access to tuples. We also design an algorithm that can optimally schedule batches, that satisfy particular properties, in quasi-linear time. We then implement it in a system called Cascade, and provide a preliminary evaluation against an existing system.

***Index Terms*—transactions, high-contention, scheduling**

I. Introduction

The study of multicore transactional systems has been, and continues to be, a well studied topic within the database community [1]–[9]. The goal of these systems is to concurrently process as many incoming transactions as possible, while maintaining the illusion that they were executed serially. Broadly speaking, we can categorize the methods for achieving this into concurrency control (CC)-based [1], [4], [5], [8] and partitioning-based [2], [3], [6], [7], [9].

The CC-based approaches, which are the most prevalent, focus on resolving conflicts between transactions as they are being executed. This can be achieved by using locking protocols such as two-phase locking (2PL), or by using timestamp or version-based protocols as present in Silo [1], TicToc [4], or Cicada [5]. On the other hand, partitioning-based approaches such as LADS [3], STRIFE [7], and TSkd [9], will partition incoming batches of transactions into smaller sets of transactions that do not contain conflicting read or write operations to the same record. These sets are then delegated to an individual thread in order to process them serially, without incurring additional performance cost for explicit coordination.

While partition-based approaches perform well for workloads that are amenable to partitioning, in the worst case, all transactions are assigned to the same disjoint set. This causes the system to execute an entire batch serially, or it will fall back to a CC-based approach to process them in parallel. In effect, paying the additional cost for up-front analysis of a batch without being able to reap the full benefits from it.

In this work, we focus on precisely those problematic batches of transactions. We can characterize such workloads as being high-contention. Indeed, when transactions share a set in the partition, it must mean that they all write to the same (often tiny) subset of tuples. To this end, we want to investigate which execution strategy is the best we could hope for, given the dire cicumstances. And perhaps more importantly, wether we can approach that execution strategy in practice.

Problem Statement. Given a batch of transactions which highly contend on writes to the same tuples, can we exploit ahead-of-execution analysis towards any beneficial outcome? More concretely, we want an answer to the following:

1) What does the optimal execution of a batch look like in these high-contention scenarios?
2) Can we come up with an algorithm that constructs schedules for a batch which correspond with their optimal execution?
3) How does it compare to existing approaches?

Contributions. We introduce a notion of optimality for transaction schedules. Intuitively, we say that a schedule is optimal if the overall time spent waiting for exclusive access to tuples is the minimal possible time (more detail in Section III). Furthermore, we provide an algorithm that can compute such optimal schedules in quasilinear time for sets of transactions $\mathcal{T}$, which have a special property we call *uniquely contending* (definition in Section V). Lastly, we provide a (preliminary) implementation of our approach (*c.f.*, Sections IV and V), and evaluate it empirically with respect to other systems (*c.f.*, Section VI).

II. Preliminaries

A. Transactions

In what follows, we will make an abstraction of tuples as *objects* within a set Obj. These objects support *read* and *write* operations, which we respectively denote by $\mathtt{R}[t]$ and $\mathtt{W}[t]$ for some $t \in \text{Obj}$. A *transaction* (over Obj) is a sequence of these read or write operations (*e.g.*, $T = \mathtt{R}[t_1], \mathtt{W}[t_2], \ldots, \mathtt{W}[t_n]$).

We model batches of incoming transactions as sets $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$. Often, we use the shorthand $\mathtt{R}_i[t]$ or $\mathtt{W}_i[t]$ to refer to operations $\mathtt{R}[t]$ or $\mathtt{W}[t]$ in some transaction T_i.

B. Schedules

A *(parallel) schedule* is a partial ordering of all operations present in some transaction set $\mathcal{T}$. When two operations from different transactions do not have a defined order with respect to one another, they are conceptually executing concurrently. Schedules are considered to be *serial* when there are no concurrent operations, and all operations in a transaction are executed contiguously, one after the other. We represent schedules graphically (*e.g.*, Figure 1).

$$\begin{array}{lllll} T_1: & \mathtt{R}_1[t] & \mathtt{R}_1[u] & \mathtt{W}_1[z] & \\ T_2: & \mathtt{R}_2[t] & & \mathtt{R}_2[u] & \mathtt{W}_2[z] \end{array}$$

$$\begin{array}{lllllll} T_1: & \mathtt{R}_1[t] & \mathtt{R}_1[u] & \mathtt{W}_1[z] & & & \\ T_2: & & & & \mathtt{R}_2[t] & \mathtt{R}_2[u] & \mathtt{W}_2[z] \end{array}$$

Fig. 1. A (parallel) schedule and a serial schedule for the set of transactions $\mathcal{T} = \{T_1 = \mathtt{R}[t], \mathtt{R}[u], \mathtt{W}[z]; T_2 = \mathtt{R}[t], \mathtt{R}[u], \mathtt{W}[z]\}$.

In the above figure, we assume that the axis of time moves from left to right. Hence, in the top schedule, the operation $\mathtt{W}_1[z]$ precedes $\mathtt{W}_2[z]$. Furthermore, the operations $\mathtt{R}_1[t]$ and $\mathtt{R}_2[t]$ appear directly above each other, this indicates that they are concurrently executing. Note the absence of a commit operation, as we will assume commits coincide with the final operation in each transaction.

C. Serializability

Two operations b_i and a_j, from distinct transactions T_i and T_j, are *conflicting* if they are operations over the same object and at least one of them is a write. If b_i and a_j are conflicting, and b_i precedes or is concurrent with a_j in a schedule s, we say that a_j *depends on* b_i in s, which we write as $a_j \rightarrow_s b_i$.

Given two schedules s and s' over the same transactions $\mathcal{T}$, we say s is *(conflict) equivalent* to s' whenever $a_j \rightarrow_s b_i$ if, and only if $a_j \rightarrow_{s'} b_i$, for all conflicting operations a_j and b_i. And, a parallel schedule is *(conflict) serializable* whenever it is conflict equivalent to a serial schedule.

We can determine if a parallel schedule is serializable by first representing all of its dependencies in a directed graph; the schedule is serializable if, and only if that graph is acyclic.

Example 1. The two schedules in Figure 1 are conflict equivalent. This is because $\mathtt{W}_1[z]$ and $\mathtt{W}_2[t]$ are the only conflicting operations, and $\mathtt{W}_2[z]$ depends on $\mathtt{W}_1[z]$ in both the parallel and serial schedule. Since the parallel schedule is conflict equivalent to the serial one, we know it is serializable. Also notice that the graph formed by the dependencies is acyclic.

III. Optimal Schedules

In this section, we define a possible notion for the optimal scheduling of transaction batches. Given a schedule s for a set of transactions $\mathcal{T}$, we define the *precedence graph* for s as a directed graph $G_s = (V, A)$ such that:

- there is a vertex in V for every operation in $\mathcal{T}$;
- for all transactions $T = a_1, a_2, \ldots, a_n$ in $\mathcal{T}$ there are arcs $(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$ in A; and
- for all non-concurrent conflicting operations b_i, a_j in s where b_i precedes a_j, there is an arc (b_i, a_j) in A.

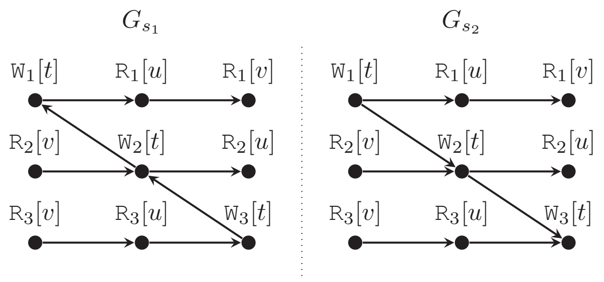
The *cost* $c(G_s)$ associated with a given precedence graph G_s is defined as the length of its longest directed path. Notice that, when a schedule is serializable, its precedence graph is a directed acyclic graph. Hence, for any serializable schedule s we can quickly compute $c(G_s)$ by doing a topological sort of G_s in linear time. Going forwards, we will only consider serializable schedules.

Example 2. Given the following two schedules s_1 and s_2,

$$\begin{array}{llllllllll} T_1: & & & & & & & \mathtt{W}_1[t] & \mathtt{R}_1[u] & \mathtt{R}_1[v] \\ T_2: & & & & \mathtt{R}_2[v] & \mathtt{W}_2[t] & \mathtt{R}_2[u] & & & \\ T_3: & \mathtt{R}_3[v] & \mathtt{R}_3[u] & \mathtt{W}_3[t] & & & & & & \end{array}$$
$$\boxed{s_1}\longrightarrow$$

$$\begin{array}{llll} T_1: & \mathtt{W}_1[t] & \mathtt{R}_1[u] & \mathtt{R}_1[v] \\ T_2: & \mathtt{R}_2[v] & \mathtt{W}_2[t] & \mathtt{R}_2[u] \\ T_3: & \mathtt{R}_3[v] & \mathtt{R}_3[u] & \mathtt{W}_3[t] \end{array}$$
$$\boxed{s_2}\longrightarrow$$

we get two associated precedence graphs G_{s_1} and G_{s_2},



Notice that $c(G_{s_1}) = 6$, while $c(G_{s_2}) = 2$. Hence, the preferred schedule is s_2 in this scenario.

At an intuitive level, a precedence graph represents all of the possible execution paths that a given schedule allows for. We can think of executing one operation, or visiting one vertex, as spending one time unit. Cross-transaction arcs can be viewed as hand-off points where exclusive access is transfered to a different thread of execution. Hence, in the worst case, we will have to traverse the longest possible path in this graph. This leads to the following definition.

Definition 1. A serializable schedule s is optimal for a set of transactions $\mathcal{T}$ if, compared to all other serializable schedules for $\mathcal{T}$, it has the minimal $c(G_s)$.

IV. Architecture

Our strategy for computing these optimal schedules is implemented in a barebones database system called Cascade, written in Rust[1]. It supports transactions on top of a concurrent key-value map data structure in main-memory.

[1]https://www.rust-lang.org/

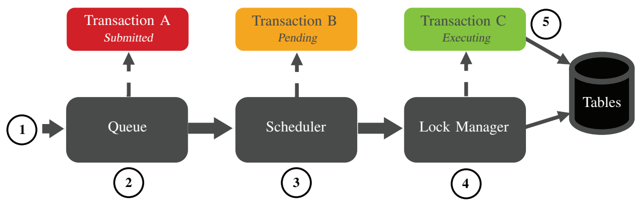


Fig. 2. An overview of Cascade's transaction pipeline.

We make a number of simplifying assumptions that are also common in the literature [3], [6], [7], [9]. That is, transactions run in one-shot mode, meaning that they will never interact with their caller once they have been submitted. And, we assume that the read and write sets are known beforehand, which implies that we cannot do reads or writes that depend on the outcome of other reads.

Since Cascade performs ahead-of-execution analysis, just as partitioning systems do, we cannot assign an operating system (OS) thread to each transaction. This would imply that we cannot do other useful work while our transaction is waiting to be assigned to a schedule. Therefore, Cascade is implemented on top of the asynchronous runtime called Tokio[2]. This means that transactions are implemented as futures, which for the purposes of this paper, you can regard as lightweight threads that are cheap to schedule and deschedule. This enables the OS thread to which the transaction is assigned to perform other work while waiting, such as computing and assigning schedules to batches, and then executing those transactions.

We use a table data structure which is backed by a concurrent B+-Tree with epoch-based reclamation[3], and each tuple is guarded by a reader-writer lock from the `parking-lot` library[4]. We make use of Rust's default memory allocator.

The architecture of Cascade is composed from asynchronous building blocks, akin to actors. Together they form a pipeline, where one component sends transaction batches downstream to the next. Each batch moves forwards through this stream by the princple of backpressure, that is, each downstream component will request more batches from its upstream components as space becomes available. This prevents overflowing a component with more than it can handle. An overview of these components can be seen in Figure 2. We will now discuss each of them one by one.

(1) Request Handling. Transactions are futures that contain a Rust closure with the database as a parameter. As it enters the system, it is first executed against an empty version of the database. This allows the system to capture the sequence of reads and writes. It is subsequently pushed into the queue, if it has space left, aborting immediately otherwise.

(2) Queue. Incoming transactions are buffered in a queue until the scheduler sends a request for more batches. The size of this queue is a constant determined by the available memory in the system. Once the transaction is pushed into the queue it goes into a *submitted* state.

(3) Scheduler. This component takes in batches of a pre-specified size, usually as big as there are available threads (minus the ones used for background tasks), and applies a scheduling algorithm to them. Its output is an ordering of the transactions. Transactions currently in this component are in a *pending* state. The specific details and algorithm are the subject of Section V.

(4) Lock Manager. Once the batch enters this component, the transactions are all assigned to a thread and executed simultaneously. The order on the transactions given as input is used to centrally manage exclusive access to certain key tuples, as a background task. Exclusive access to non-centrally managed tuples is handled by the transaction itself. Exactly which tuples are centrally managed is discussed in Section V and VII.

(5) Transaction Execution. During execution, some operations may access centrally managed tuples. At such a point, it will block until the lock manager gives it exclusive access to that record. For other tuples, it will obtain exclusive access on its own. The key idea being that the time spent waiting will be bounded by the order in which the lock manager provides exclusive access.

V. Design

A. Uniquely Contending Transactions

Our scheduling algorithm works for a specific sets of transactions we characterize in the following way.

Definition 2. A set of transactions $\mathcal{T}$ is said to *uniquely contend* whenever:

- there is precisely one object u accessed by more than one transaction in $\mathcal{T}$, and
- each transaction writes to u exactly once.

Given a setting like this, we can think of each transaction $T \in \mathcal{T}$ as a sequence

$$T : \underbrace{\mathtt{W}[t_1]\ \mathtt{W}[t_2]\ \dots}_{\text{prefix}}\ \mathtt{W}[u]\ \underbrace{\dots\ \mathtt{W}[t_{n-1}]\ \mathtt{W}[t_n]}_{\text{suffix}}$$

where T has a (potentially empty) subsequence of non-conflicting writes, followed by a write to u, and another subsequence of non-conflicting writes. The former subsequence is refered to as the *prefix* of T; the latter is the *suffix* of T. We often refer to the number of writes in the prefix and suffix as $\mathit{prelen}(T)$ and $\mathit{suflen}(T)$ respectively.

As mentioned in Section IV, the lock manager will centrally take care of exclusive access to certain tuples. In the case of uniquely contending transactions, it will be responsible for the unique object u, on which every transaction conflicts. For all other objects, the transactions manage their exclusive accesses individually, since they never block.

[2]https://tokio.rs/
[3]https://github.com/komora-io/concurrent-map
[4]https://github.com/Amanieu/parking_lot

B. Allocations

The main construct in our algorithm is an *allocation*. Conceptually, an allocation assigns all transactions in some set of uniquely contending transactions to one of $k+1$ possible bins, depending on their prefix and suffix length. In our implementation, they are efficiently represented as fixed arrays of length $k+1$.

Definition 3. Given a positive integer k, and $\mathcal{T}$ a set of uniquely contending transactions, an *allocation* to $k+1$ bins is a one-to-one function $alloc_{k+1} : \mathcal{T} \to \{0, 1, 2, \ldots, k\}$ where:

- $prelen(T) \leq alloc_{k+1}(T)$; and
- $suflen(T) \leq k - alloc_{k+1}(T)$.

There is a natural correspondence between allocations and schedules. Namely, if two transactions T_1 and T_2 satisfy $alloc_{k+1}(T_1) < alloc_{k+1}(T_2)$, then T_1 is ordered before T_2 in its matching schedule. We write that schedule as $s(alloc_{k+1})$.

Allocations provide us with an important benefit over schedules, since they define an upper bound on the worst-case length of a path in their corresponding precedence graph $G_{s(alloc_{k+1})}$. We demonstrate this connection in the following proposition.

Proposition 1. *Let $\mathcal{T}$ be any set of uniquely contending transactions. If there is an allocation $alloc_{k+1}$ for $\mathcal{T}$, then the precedence graph G of $s(alloc_{k+1})$ satisfies $c(G) \leq k+1$.*

Proof. Let p be an arbitrary path in G which starts in in the i'th transaction T_i of $s(alloc_{k+1})$ and ends in its j'th transaction T_j. The number of operations in p is $prelen(T_i) + j - i + 1 + suflen(T_j)$. First, observe that this is bounded above by:

$$alloc_{k+1}(T_i) + j - i + 1 + k - alloc_{k+1}(T_j)$$

This follows from the fact that $prelen(T_i) \leq alloc_{k+1}(T_i)$ and $suflen(T_j) \leq k - alloc_{k+1}(T_j)$. Next, notice that:

$$\begin{aligned} & alloc_{k+1}(T_i) + j - i + 1 + k - alloc_{k+1}(T_j) && \leq \\ & alloc_{k+1}(T_i) + alloc_{k+1}(T_j) - alloc_{k+1}(T_i) + \\ & 1 + k - alloc_{k+1}(T_j) && \leq \\ & k+1, \end{aligned}$$

since $j - i \leq alloc_{k+1}(T_j) - alloc_{k+1}(T_i)$. Because p was arbitrary, this shows that every p has at most $k+1$ operations, which implies $c(G) \leq k+1$. □

C. The Scheduling Algorithm

Now that allocations give us a way to figure out the worst cost for a given ordering, we still have to construct them or show we cannot. Hence, we present $\text{ALLOC}(\mathcal{T}, k)$ as seen in Algorithm 1, which returns a allocation with $k+1$ bins for $\mathcal{T}$, or $\perp$ if no allocation exists for $\mathcal{T}$.

We then call Algorithm 1 in a loop to determine the optimal path length k. This should minimally be $|\mathcal{T}|$ long, hence why we start iterating from $|\mathcal{T}|-1$. This is captured in Algorithm 2.

Observe that Algorithm 1 involves sorting all transactions, which can be done in $|\mathcal{T}| \log |\mathcal{T}|$ time, in the worst case. All other steps involve 2 linear passes, which is $2|\mathcal{T}|$. Its overall time complexity is $O(2|\mathcal{T}| + |\mathcal{T}| \log |\mathcal{T}|) = O(|\mathcal{T}| \log |\mathcal{T}|)$.

Algorithm 1: $\text{ALLOC}(\mathcal{T}, k)$

```
Input: uniquely contending transactions $\mathcal{T}$, and $k \geq 1$
Output: $alloc_{k+1}$ for $\mathcal{T}$ or $\perp$
1 array[$k+1$] $a$ = new array[$k+1$];
2 stack[$k+1$] $s$ = stack from sorting $\mathcal{T}$ by $T_a < T_b$ if
     $suflen(T_a) > suflen(T_b)$, and
     $prelen(T_a) < prelen(T_b)$ if $suflen(T_a) = suflen(T_b)$;
3 while $s$ is not empty do
4     txn $T$ = $s$.pop();
5     int $j$ = find the minimal free index $i$ of $a$ with
        $prelen(T) \leq i$ and $suflen(T) \leq k - i$;
6     if $j$ does not exist then
7         return $\perp$;
8     $a[j]$ = $T$;
9 return $a$;
```

Algorithm 2: $\text{SCHEDULE}(\mathcal{T})$

```
Input: uniquely contending transactions $\mathcal{T}$
Output: $alloc_{k+1}$ for $\mathcal{T}$
$k$ = $|\mathcal{T}| - 1$;
while ALLOC($\mathcal{T}, k$) is $\perp$ do
    $k$ += 1;
return ALLOC($\mathcal{T}, k$);
```

For Algorithm 2, we repeat Algorithm 1 a constant number of times. Namely, between $|\mathcal{T}|-1$ and $|\mathcal{T}|-1$ plus the length of the longest transaction in $\mathcal{T}$. Verification of the soundness and completeness of the above algorithms is omitted in the interest of space.

VI. Evaluation

These evaluation results are preliminary; they should not be taken as a comprehensive evaluation of this approach. We discuss our plans in the future work section.

A. Experimental Setup

System. The following experiments were conducted on a single system running Linux 5.15.133 with a AMD Ryzen 9 7950X processor at a base clock of 4.50 GHz (with a maximal boost clock up to 5.70 GHz). The system has access to 32 logical cores and 16 physical cores. Each core has access to a 64 KB L1-cache, and a 1 MB L2-cache. And, all cores share a 64 MB L3-cache. The system has access to 32 GB of DRAM.

Basis Factors. In the upcoming benchmark we evaluate our approach against the same B+tree implementation, the same allocator, and the same workload. The only factors that are different pertain the the concurrency control approach itself. We test our implementation against the optimistic concurrency control (OCC) protocol Silo [1], which does not execute on an asynchronous runtime, but instead runs on OS threads. Furthermore, it uses timestamping to regulate access to tuples, instead of reader-writer locks.

B. Preliminary Experiment

We subjected both our implementation of the OCC protocol Silo [1] and Cascade to an end-to-end uniquely contending transaction workload. This implies that there is 100% contention for a single tuple, as each transaction will conflict with every other transaction. Each transaction has a randomised suffix and prefix length between $[0, 10]$ operations. The results of this experiment are shown in Figure 3.

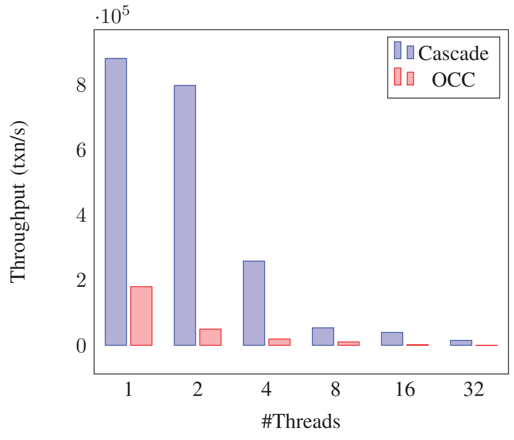


Fig. 3. Median throughput for a uniquely contending transaction workload. The reported figures are transactions processed per second as the logical core count increases from 1 to 32, in powers of two.

VII. Future Work

A. Algorithm Extensions

The current scheduling algorithm can only handle a limited class of transaction sets that are uniquely contending. A good step forwards could be to investigate in which dimensions the algorithm can be extended while remaining tractable. For instance, can we introduce two or more highly accessed tuples at once? Maybe there are certain constraints on the distribution of those tuples which make sense to consider.

Of course, there is always a possibility that even the slightest extension makes the algorithm infeasible in practice. Therefore, a second orthogonal approach would be to not consider optimality for all possible classes. Instead we could focus on providing a heuristic-based approach with our current algorithm as a starting point.

B. Benchmarks

Cascade is currently severely under-benchmarked. The reason for this being that implementation of existing transaction protocols on top of the same basis factors is time consuming to do as a single Ph.D. student. However, I do want to outline which benchmark plans are in the pipeline in the hopes of receiving feedback on whether they are in the right direction.

First, I want to implement two-phase locking as the second concurrency-control based approach since the unique contention benchmark heavily favors pessimistic approaches. Secondly, I intend to add a wider range of benchmarks with lower amounts of contention; since 100% contention should rarely occur in practice. And finally, there have to be way more micro-benchmarks to determine which aspects of the system prove to be bottlenecks. In particular, some parts of the current implementation use more memory than they should.

C. Speculative Scheduling

A major assumption of the current work is that read and write sets are known beforehand. In many practical workloads, reads and writes can act on the result of other reads and writes. Transactions can additionally contain conditional structures or even loops which act on the output of a given read. In our approach we cannot allow any of those transactions.

One possible approach could be to predict the output of certain reads and writes based on previous reads and writes to that object. We could then use the most-queried object as a proxy for which object will be returned from that read. In the worst case, this object will not match up with reality, but then we just have a slightly worse than optimal schedule; this should not impact serializability of the generated schedule.

References

[1] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 18–32. [Online]. Available: https://doi.org/10.1145/2517349.2522713

[2] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1190–1201, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1190-faleiro.pdf

[3] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W. Wong, and M. Zhang, "Exploiting single-threaded model in multi-core in-memory systems," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 10, pp. 2635–2650, 2016. [Online]. Available: https://doi.org/10.1109/TKDE.2016.2578319

[4] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1629–1642. [Online]. Available: https://doi.org/10.1145/2882903.2882935

[5] 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, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 21–35. [Online]. Available: https://doi.org/10.1145/3035918.3064015

[6] J. M. Faleiro, D. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 613–624, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p613-faleiro.pdf

[7] G. Prasaad, A. Cheung, and D. Suciu, "Handling highly contended OLTP workloads using fast dynamic partitioning," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 527–542. [Online]. Available: https://doi.org/10.1145/3318464.3389764

[8] Y. Huang, W. Qian, E. Kohler, B. Liskov, and L. Shrira, "Opportunities for optimism in contended main-memory multicore transactions," *VLDB J.*, vol. 31, no. 6, pp. 1239–1261, 2022. [Online]. Available: https://doi.org/10.1007/s00778-021-00719-9

[9] Y. Cao, W. Fan, W. Ou, R. Xie, and W. Zhao, "Transaction scheduling: From conflicts to runtime conflicts," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 26:1–26:26, 2023. [Online]. Available: https://doi.org/10.1145/3588706