An Evaluation of Strict Timestamp Ordering Concurrency Control for Main-Memory Database Systems

Stephan Wolf, Henrik Mühe, Alfons Kemper, Thomas Neumann

Technische Universität München
{wolfst,muehe,kemper,neumann}@in.tum.de

Abstract: With the fundamental change of hardware technology, main-memory database systems have emerged as the next generation of DBMS. Thus, new methods to execute transactions in a serial, lock-free mode have been investigated and successfully employed, for instance in H-Store or HyPer. Although these techniques allow for unprecedentedly high throughput for suitable workloads, their throughput quickly diminishes once unsuitable transactions, for instance those crossing partition borders, are encountered. Still, little research concentrates on the overdue re-evaluation of traditional techniques, that do not rely on partitioning.

This paper studies strict timestamp ordering (STO), a "good old" technique, in the context of modern main-memory database systems built on commodity hardware with high memory capacities. We show that its traditional main drawback – slowing down reads – has a much lower impact in a main-memory setting than in traditional disk-based DBMS. As a result, STO is a competitive concurrency control method which outperforms the partitioned execution approach, for example in the TPC-C benchmark, as soon as a certain percentage of the workload crosses partition boundaries.

1 Introduction

In recent years, hardware with large capacities of main memory has become available, leading to a renewed interest in main-memory database systems. Here, page faults no longer need to be compensated by executing parallel transactions, which allows for removing many synchronization components that are indispensable in traditional, disk-based database systems. Harizopoulos et al. [HAMS08] found, that most time spent executing a transaction is actually used by components like buffer manager, lock manager and latching.

Without the need for hiding I/O latencies, other execution paradigms like partitioned serial execution, as first investigated by Kallman et al. [KKN$^+$08] in their H-Store prototype, become viable alternatives to traditional locking. Here, transactions are executed sequentially on each partition of the data without the need for any concurrency control at all.

Even though a sequential execution approach leads to outstanding performance when the data and workload allow for partitioning in a suitable way [KKN$^+$08, KN11], partition crossing transactions quickly lead to a deterioration in throughput, even on a single node without additional network delays (see Figure 1). One reason is that current implementa-

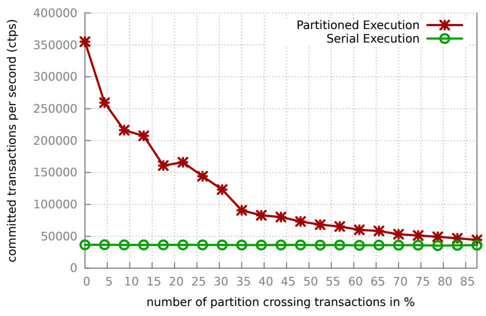

Figure 1: Throughput decrease related to the ratio of part partition-crossing transactions.

tions often rely on coarse granularity synchronization mechanisms, like the full database lock used in the HyPer DBMS prototype [KN11].

In this paper, we reinvestigate the "good old" timestamp-based concurrency control as suggested in [BHG87, Car83] decades ago. Major drawbacks of the timestamp approach – like having to write a timestamp for every read – have to be re-evaluated when data resides in main-memory.

The remainder of this paper is structured as follows: In Section 2, we will introduce both partitioned serial execution, as well as the strict timestamp ordering approach (STO) evaluated in this work. Section 3 describes our implementation of STO inside the HyPer database system prototype and highlights the most severe adjustments required when using timestamp-based concurrency control mechanisms. We offer a thorough evaluation of STO, as well as a comparison of STO with partitioned serial execution in Section 4. Section 5 concludes this paper.

2 Formal Background

Before discussing the implementation of strict timestamp ordering in HyPer, we will provide the theoretical background of the algorithm. Additionally, we outline serial execution and partitioned execution, which we will compare to strict timestamp ordering.

2.1 Serial Execution

Traditional disk-based database systems frequently rely on locking to achieve serializability among concurrent transactions. When reading or writing data to disk, this is essential since I/O latency need to be masked. In main-memory database systems, however, the need for masking I/O misses no longer exists allowing for the efficient serial execution of suitable workloads without traditional concurrency control.

H-Store [KKN$^+$08] pioneered the idea of removing buffer management, as well as locking and latching from main-memory database systems, allowing for the efficient execution of partitionable workloads with minimal overhead. This concept, which we refer to as *serial execution*, has since been picked up by other main-memory database systems, for instance the commercialized version of H-Store named VoltDB [Vol10] as well as our HyPer research prototype DBMS [KN11].

Unlike VoltDB, HyPer also supports mixed OLTP/OLAP applications by separating the two disparate workloads using virtual memory snapshotting [MKN11]. Here, we concentrate only on the OLTP synchronization.

2.2 Partitioned Execution

Scaling the transactional throughput when using serial execution is possible by running multiple serial execution threads in parallel for disjoint partitions of the data. As shown by Curino et al. [CZJM10], some workloads can be partitioned such that cases where a transaction has to access multiple partitions are rare. For the TPC-C benchmark[1], for instance, only 12.5% of all transactions access more than one partition of the data.

Other main memory database systems, which rely on partitioning, disallow the execution of transactions which might access more than one partition of the data. In contrast, HyPer executes transactions assuming that they will operate on only one data partition. If a transaction accesses data outside its own partition, a database lock is acquired causing transactional processing to fall back into serial execution mode without concurrency on separate partitions. After the transaction has finished, the database lock is released and concurrent execution on all partitions of the database is resumed. We call this execution mode *partitioned execution* or *PE* for short.

2.3 Strict Timestamp Ordering (STO)

Timestamp-based concurrency control uses timestamps for synchronization instead of locks. From the outside it seems that the transactions are executed sequentially according to their starting time. In other words, the scheduler generates serializable schedules that are equal to the serial execution of the transactions ordered by their starting time.

To achieve this, the transaction manager assigns a timestamp $TS(T_i)$ to each transaction T_i at its start and guarantees that the timestamp of transactions that started later is always higher than the timestamps of all earlier transactions. These timestamps are used to guarantee the *Timestamp Ordering (TO)* rule: if two operations $p_i(x)$ and $q_j(x)$ are in conflict, i.e. they access the same tuple x and at least one operation is a write operation, then the operation of the transaction with the lower timestamp is always executed first. Thereby, the resulting schedule is equal to the serial execution of the transactions ordered by their timestamp and, as a consequence, it is serializable.

[1]See `http://www.tpc.org/tpcc/`

In order to enforce the TO rule, the database system has to save the timestamp of the transaction which has last read tuple x, and the timestamp of the transaction which has last changed tuple x. In the following, these timestamps are denoted as $readTS(x)$ and $writeTS(x)$.

With these meta data, the transaction manager is able to perform the following test, which enforces the TO rule:

1. $r_i(x)$: T_i wants to read x:
   (a) If $TS(T_i) < writeTS(x)$, the TO rule would be violated. Thus, the transaction T_i has to be aborted.
   (b) Otherwise, allow access and set
   $readTS(x) := max(TS(T_i), readTS(x))$.
2. $w_i(x)$: T_i wants to write x:
   (a) If $TS(T_i) < readTS(x)$ or $TS(T_i) < writeTS(x)$,
   the TO rule would be violated. Thus, the transaction T_i has to be aborted.
   (b) Otherwise, allow access and set $writeTS(A) := TS(T_i)$.

It is required that write operations on the same data tuple are executed atomically and write and read operations are mutually excluded.

According to [BHG87] and [CS84], this algorithm is called *Basic Timestamp Ordering (BTO)*. It generates serializable schedules, but does not guarantee recoverability. In fact, aborted transactions can cause inconsistency, as another transaction which accessed dirty data could already be committed.

As recoverability is essential for database systems, we employ an extension of Basic Timestamp Ordering called *Strict Timestamp Ordering (STO)* [BHG87]. STO does not only provide recoverable schedules, but also strict schedules. That means, that no uncommitted changes of a running transaction are overwritten or read by another transaction. This is prevented by the use of a dirty bit. Each transaction marks tuples with uncommitted changes by setting the dirty bit and other transactions accessing such a tuple have to wait until the dirty bit is unset, which happens when the previous transaction commits or is rolled back. In order to avoid deadlocks, the transaction manager has to ensure that a transaction never waits for younger transactions. Thereby, cyclic waiting is prevented, which is one of the necessary Coffman conditions for a deadlock [CES71].

3 Implementation of STO

In order to evaluate the performance of STO, we used the database system HyPer [KN11] to implement the described algorithm. HyPer is an in-memory, high-performance hybrid OLTP and OLAP DBMS that originally relies on sequential execution for transaction processing. To further improve transaction processing throughput, transactions are not interpreted but are compiled to machine code using the LLVM compiler back-end. This removes interpretation overhead at runtime and improves hardware optimizations, for example branch prediction [Neu11].

The implementation of STO in HyPer required not only a new transaction manager, but also architectural modifications because of concurrency inside partitions. These impacts of STO on the architecture will be described in Section 3.2. Before that, the basic data structures needed by the STO implementation are presented to provide a better understanding of the implementation.

3.1 Data Structures

Besides the read and write timestamps, further data structures were necessary. To avoid that dirty data is read or overwritten, a dirty bit is needed. Furthermore, because of reasons presented in Section 3.2.1, our implementation requires a delete flag. And last but not least, a dirty bit inventory was needed, which is responsible for unsetting the dirty bits after a transaction has aborted or committed.

3.1.1 Timestamp Codes

We used 32-bit values for the read and write timestamps and encoded the dirty bit and delete flag into the write timestamp. The highest bit is reserved for the dirty bit and the delete flag is set when all other 31 bits of the write timestamp are set. This design has two advantages compared to a separate delete flag and dirty bit:

- As the write timestamp has to be checked anyway, the check for the dirty bit does not require an additional memory operation. Furthermore, checking if the dirty bit is not set and the write timestamp is lower than the transaction's timestamp requires only one arithmetic operation.
- The delete flag design is beneficial, as it makes a separate check for tuple deletion unnecessary. When the delete flag is set, the write timestamp is equal to the highest possible timestamp. So, all transactions accessing the deleted tuple will abort without an additional check of the delete flag.

As the transactions' timestamps have to be assigned in strictly increasing order, the size of the timestamp variables determines when the timestamp arrays have to be reset. If a database processes 250 000 transactions per second in a lab setting (in almost every real-world scenario, this throughput is not required), the timestamps would have to be reset only after approximately 2 hours.

This can be done as follows: When a new transaction is started and acquires a new timestamp, it is checked if the value range is exceeded. If this is the case, all running transactions are rolled back, all timestamp fields are reset, and the aborted transactions are restarted. The impact of aborting running transactions is negligible, as the length of OLTP transactions is short. For domains, where a short and rare delay during transaction processing is not tolerable, 64-bit timestamps can be used.

3.1.2 Dirty Bit Inventory

The dirty bit inventory is necessary for resetting the dirty bits and is maintained for each running transaction. Whenever a transaction sets the dirty bit of a tuple which was not set before, the tuple identifier is inserted into the transaction's dirty bit inventory. After a transaction aborts or commits, the dirty bit inventory is processed and the transaction's dirty bits are unset. As a tuple identifier is only ever inserted once into the dirty bit inventory and as each tuple identifier cannot be in two dirty bit inventories of different transactions at the same time, it need not be checked whether the dirty bit is set and originates from the current transaction, which simplifies resetting the dirty bit.

3.2 Architectural Details

By contrast to partitioned execution, strict timestamp ordering allows multiple concurrent transactions inside partitions. We will briefly discuss the necessary architectural adaption in this section.

3.2.1 Undoing Deletes

One problem is that concurrency on partition level could violate recoverability. When a transaction aborts, all its effects have to be undone. If the transaction has deleted tuples, they have to be reinserted. However, this could fail in a naive implementation because of violations of unique keys, if a concurrent transaction has inserted a tuple with the same key in the meantime.

We solved this problem by deferring the removal of tuples to the commit phase of a transaction. Deleted tuples are marked with the delete flag and the dirty bit is set, so that other transactions trying to access this tuple will wait. The deleting transaction skips this tuple the next time it tries to access it.

3.2.2 Index Structures and Synchronization

Index structures need to be refitted to support concurrent access. Optimizing index structures for concurrency is an active and complex topic of research. Transactional memory implementations [DFGG11, DGK09], as well as relativistic programming [TMW10, HW10] provide promising results on modern hardware.

In our implementation, we use full index latching to synchronize access to index structures. This is reasonable, as each partition has its own index structures. However, when done naively, this solution can constitute a major performance bottleneck as shown by [HAMS08], who analyzed the overhead of traditional locking in the context of main-memory database systems. We evaded this issue by optimizing the lock implementation. Concretely, we used an adapted version of the MCS lock [MCS91], which uses spinning on thread-local variables for waiting and allows reader and writer synchronization. That boost the performance of our STO implementation by a factor of two compared to traditional latching using the lock implementation from the `pthreads` library. Furthermore, we avoid locking the index structures for each tuple. Instead, if we subsequently access tuples from the same partition, we keep the lock until we switch to the next partition.

3.2.3 Synchronization of Admissibility Check

Besides the index structures, we also have to ensure that the check for admissibility of a transaction's operation is thread-safe. As we have to mutually exclude access to the index structures, we avoiding the necessity of additional locking, by extending this critical section to also contain the check of admissibility. Concretely, when accessing a tuple from one partition, we lock its index structures, lookup the tuple, perform the admissibility check and access the tuple. Before switching to the next partition, we release the lock, so that other transactions can proceed working on that partition. Access to the dirty bit inventory does not need to be synchronized, as each transaction has its own inventory.

4 Evaluation

In this Section, we will evaluate the strict timestamp ordering approach and compare its performance to partitioned serial execution. All benchmarks were conducted on a Dell PowerEdge R910 server with 4x Intel Xeon X7560 processors each containing eight cores clocked at 2.26GHz. The server is equipped with 1TB of main-memory split into 64x 16GB RDIMMs connected to four separate memory controllers interconnected by Intel's Quick Path Interconnect technology. For our evaluation, redo logging was disabled for all approaches, to ensure that the results are not distorted by effects resulting from the logging technique we use.

4.1 Read versus Write Performance

One reason, why STO performed poorly on disk-resident database systems, is that it significantly slowed down read operations: Updating the read timestamp caused additional disk latency. In memory resident database systems, I/O latency is not dominating the performance any more. Therefore, we re-evaluated the read performance of STO.

For this, we designed a microbenchmark. It consists of one large table with 10 million tuples. Each tuple consists of two attributes: a 64-bit integer *key* and a 64-bit integer *value*. A hash map is used as primary index. The table is divided into 128 partitions by using Fibonacci hashing on the primary key. To avoid conflicts, each thread has its own set of tuples, which we call the threads *workset*. Concretely, the first thread accesses only the first $\lfloor 10\ \mathit{million}/(\mathit{number\ of\ threads})\rfloor$ tuples, the second thread the following $\lfloor 10\ \mathit{million}/(\mathit{number\ of\ threads})\rfloor$ tuples, etc.

The benchmark offers two modes: read or write. In both modes, there is only one type

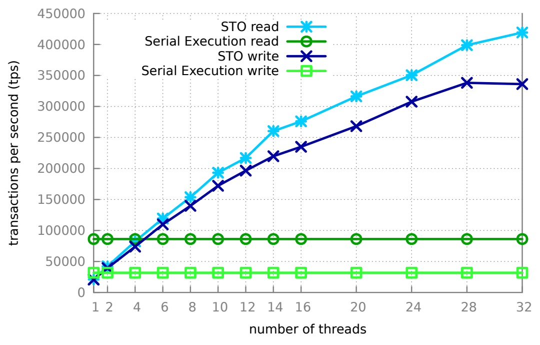

Figure 2: The read and write performance of STO

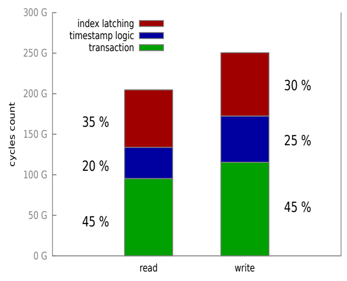

Figure 3: Cycles Distribution

of transaction that is provided with an array of 50 primary keys taken from the threads workset. In write mode, the transactions increment the *value* attribute of the corresponding tuples and in read mode, the transactions fetch the *value* attribute of each tuple and check a condition, that is always false, to avoid that the query optimizer removes the fetch query, as the value is not used.

As the partitions are arranged by the Fibonacci hash of the primary key, the workset of each thread is uniformly distributed over all partitions. This has two implications: First, all transactions are partition-crossing. Second, the transactions interfere with each other by latching the partitions' index structures. But they do not conflict, as the data sets are disjoint.

Figure 2 shows the results from the micro benchmark subject to the number of threads. The duration of processing 1 million transactions was measured and the transactions per second (tps) throughput determined. Three runs were executed for each measurement and the mean was taken.

STO's read and write curve both start nearly with the same throughput. The slope exhibits linear growth up to 16 threads. Each additional thread constantly increases the throughput by about 20 000 tps. Starting from 16 threads, the system uses hyper-threading to execute the software threads. As a result, the gradient slowly declines and the throughput increase gained by adding a new thread declines with each additional thread. Still the throughput increases at a slower rate of about 10 000 tps on average.

Looking at write performance, STO can outperform serial execution when using at least 2 threads. Furthermore, by using 32 threads, we can increase the throughput by one order of magnitude compared to the serial execution.

In read mode, STO achieves about 25% higher peak throughput than in write mode. In contrast, serial execution achieves a difference of a factor of 2.5. This shows that the traditional problem of STO – slowing down read operations – still persists in main-memory database systems but its impact is reduced: While in disk-resident database systems the difference between read and write operations was about one order of magnitude because of disk latency, in main-memory database systems, the difference is about a factor of 2 to 3.

4.2 Overhead analysis

As it was shown by Harizopoulos et al. [HAMS08] that latching in traditional database systems produces severe overhead, we employed a lock implementation that is optimized for highly parallel systems, called the MCS lock. Still, we should differentiate between the overhead produced by the STO logic and the overhead produced by latching, as STO does not rely on latching index structures. For example, lock-free index structures or index structures which rely on relativistic programming [HW10, TMW10] could be used.

We analyzed how much time is needed for each component of the concurrency control approach: Latching, STO logic and execution of the transaction itself using the previous benchmark in both modes. For determining the time difference between two evaluation points, we used the CPU cycles counter. Concretely, we defined measure points before and after each latching operation as well as each STO operation. At these points, the difference between the current cycles count and the cycles count at the previous measure point is computed and the result is added to a thread-local summation variable for each phase.

Figure 3 shows the resulting distribution taken from one run with 32 threads. Similar results were obtained when using a different number of threads and are therefore omitted here. It can be observed that the total cycles count of the write transactions is about 25% higher than of the read transaction, which matches the result from the write and read comparison.

Furthermore, in both cases, the basic transaction instructions such as updating tuples, fetching tuples, etc., cover about half of the time of a transaction. In read mode, this does not seem to fit to the previous benchmark, where serial execution was about 4 times faster than STO run with a single thread. Concretely, the time needed for the basic transaction instructions should be about one quarter of the cycles total. The reason for this difference can be explained by cache effects. When, for example, a timestamp is updated, the changes will be written into the processor's cache. As a result, the expensive propagation of the change to the main-memory will happen, when the cache line is replaced, which is usually caused by a read operation. As the basic transaction instructions are read intensive – looking up primary keys in the hash map, fetching tuples – they are likely to replace cache lines and cause costly propagation to main-memory. As a consequence, the expensive write operations caused by latching or timestamp maintenance slow down the basic transaction instructions, as these are read intensive. Therefore, half of the overhead of the basic transaction instructions seems to be also caused by locking and latching. In write mode, this effect is not significant. Here, the analysis reflects the results of the previous benchmark: When using one thread STO achieves about half of the performance of serial execution.

The overhead caused by concurrency control is distributed similarly in read and write mode. Although the STO overhead in write mode is higher than in read mode – the dirty bit inventory has to be processed and the dirty bits have to be reset – it can be observed that in both cases index latching causes more overhead than the STO logic itself. Nevertheless the optimized MCS lock could decrease the overhead of latching by about a factor of 4 compared to the results of running a disk-resident database in main-memory [HAMS08].

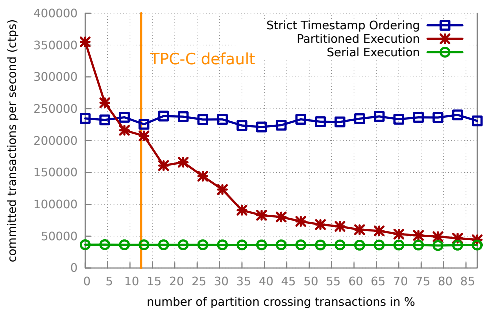

Figure 4: TPC-C benchmark with varying the number of partition-crossing transactions

Still, index latching produces significant overhead and we will investigate the performance improvements achievable with lock free index structures in future research.

4.3 Strict Timestamp Ordering versus Partitioned Execution

Finally, we compare strict timestamp ordering to partitioned execution while varying the number of partition crossing transactions. For the analysis, we used the well-known TPC-C benchmark[2] as it is easily partitionable by using the warehouse id and widely used as a benchmark for main-memory database systems comparable to HyPer, for instance in [KKN$^+$08, Vol10].

In the TPC-C benchmark there are two types of transactions that cross partition borders – 25% of the *payment* transactions and 10% of the *new order* transactions. Regarding their ratio in the total workload, this leads to a total of about 12.5% partition-crossing transactions. For this benchmark, we equally adjusted the percentage of partition-crossing *payment* and *new order* transactions from 0% to 100%, resulting in a total ratio of 0% to $87,5\%$.

In Figure 4, we show the average sustainable throughput of serial execution, partitioned execution and strict timestamp ordering while varying the percentage of transactions which cross partition boundaries. In order to provide a fair comparison, we counted only the number of committed transactions per second, as STO solves conflicts by aborting the conflicting transaction. We set the number of warehouses to 128, which resulted in about 17 GB of data, and used 20 threads.

When no partition crossing transactions are included in the workload, PE performs significantly better than STO. Here, conditions are optimal for PE as every transaction is restricted to one partition of the data and no locking is necessary at all. STO, on the other hand, requires atomic operations for locking and needs to update read/write timestamps. Therefore, the throughput achieved by STO is about 33% lower than the throughput of PE.

For an increased number of partition crossing transactions, PE's throughput declines sig-

[2]See `http://www.tpc.org/tpcc/`

nificantly. At 12.5% partition crossing transactions – the percentage in the original TPC-C – the throughput achieved by PE has already dropped below the throughput achieved with STO. As the number of partition crossing transactions increases further, the throughput curve converges to the throughput achieved by serial execution. This was to be expected, since PE uses serial execution without parallelism for partition crossing transactions causing it to behave like serial execution for high percentages of partition crossing transactions.

STO exhibits constant throughput regardless of how many transactions cross partition borders. This is due to its reliance on per-tuple timestamps which a) constitutes a fine-granularity concurrency control method and b) does not require a centralized locking infrastructure. Thus, it is perfectly suited for workloads that can not be completely partitioned.

5 Conclusion

In this paper, we re-evaluated the traditional strict timestamp ordering concurrency control algorithm in a main-memory database system on modern hardware, while most modern main-memory DBMS omit explicit concurrency control in favor of partitioning and serial execution.

We found that the traditional drawback of STO – slowing down read operations as if they were write operations – is less significant in main-memory than in disk-based database systems. Here, the performance of read and write operations differs by about a factor of 2, whereas in disk-resident database systems the difference was at least one order of magnitude because of disk latency.

As a result, STO is a competitive alternative to partitioned execution: While partitioned execution is – by design – ideal for a perfectly partitionable workload, STO allows the efficient execution of workloads regardless of the quality of the underlying partitioning. Even a low number of partition-crossing transactions, for example the default ratio of 12.5% partition crossing transactions in the TPC-C benchmark, suffice that STO outperforms PE. Therefore, STO is suitable for environments where transactions can not be easily restricted to work on only one partition of the data.

Additionally, we found that traditional bottlenecks like latching need to be re-evaluated from an implementation standpoint: We could improve the performance of STO by a factor of 2 by using an optimized latch implementation which uses thread-local spinning. Still, the overhead of latching stays a significant factor and it should be evaluated if technologies like lock-free index structures, transactional memory or relativistic programming can further reduce it.

In summary, re-investigating the suitability of traditional works in concurrency control for their performance in a fundamentally changed hardware environment has allowed us to find a more robust concurrency control method for main memory DBMS that is competitive to current approaches.

References

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, New York, 1987.

[Car83] Michael James Carey. *Modeling and Evaluation of Database Concurrency Control Algorithms*. PhD thesis, University of California, Berkeley, 1983.

[CES71] E.G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[CS84] M.J. Carey and M. Stonebraker. The Performance of Concurrency Control Algorithms for Database Management Systems. In *Proceedings of the 10th International Conference on Very Large Data Bases*, VLDB '84, pages 107–118, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[CZJM10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[DFGG11] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a Research Toy. *Communications of the ACM*, 54:70–77, April 2011.

[DGK09] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching Transactional Memory. *ACM SIGPLAN Notices*, 44:155–165, May 2009.

[HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 981–992, New York, New York, USA, 2008. ACM Press.

[HW10] Philip W. Howard and Jonathan Walpole. Relativistic Red-Black Trees. Technical report, PSU Computer Science Department, Portland, Oregon, USA, 2010.

[KKN$^+$08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[MCS91] J.M. Mellor-Crummey and M.L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[MKN11] Henrik Mühe, Alfons Kemper, and Thomas Neumann. How to Efficiently Snapshot Transactional Data: Hardware or Software Controlled? In *DaMoN*, pages 17–26, 2011.

[Neu11] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[TMW10] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Scalable Concurrent Hash Tables via Relativistic Programming. *ACM SIGOPS Operating Systems Review*, 44(3):102–109, August 2010.

[Vol10] VoltDB LLC. VoltDB Technical Overview. `http://voltdb.com/_pdf/VoltDBTechnicalOverviewWhitePaper.pdf`, 2010.