Performance Study on an In-Memory OLTP Database

Huanchen Zhang, Zhuo Chen

School of Computer Science, Carnegie Mellon University

1. INTRODUCTION

OLTP (online transaction processing) database systems are essential building blocks for many popular web services (e.g. Amazon) [13]. Traditionally, OLTP systems are no different than other database management systems (DBMS), as they all use some popular and general relational DBMS, such as DB2 [9] and Shore [1]. In the past few years, however, a number of multicore main-memory OLTP databases, including Hekaton [6], VoltDB [4], HStore [10] and Silo [15], have emerged and have been gaining more and more attention due to their extraordinary performance. For example, Silo can achieve 700,000 transactions per second running TPC-C benchmark on a 32-core machine [15], which is a 2-order-of-magnitude improvement over disk-based relational DBMS. Silo-R, a recent extension of Silo, adds durability to the database and can also achieve 550,000 transactions per second under the same settings [16].

Pushes from 2 aspects contribute to this technical transition. First, OLTP market characteristics make main memory a desirable choice. OLTP market usually deals with business data processing [7], which requires high peak throughput. For example, China's e-commerce giant Alibaba receives 278 million orders ($9.3 billion worth) during a 24-hour online shopping festival on November 11th, 2014 [14], with its 1st billion worth of orders placed in less than 20 minutes [7]. Such high throughput might be challenging for current disk-based databases to match. On the other hand, main memory has become sufficiently cheap (and faster) to host OLTP datasets. A modern commercial server typically has several terabytes of RAM [15], which is more than enough for most OLTP workloads. Moreover, the size of OLTP systems usually do not scale exponentially as RAM capacity does, because customer and real world entities do not obey Moore's law [8].

Seven years ago, Harizopoulos et. al. presented a performance breakdown graph running Shore in memory [8]. In this work, we conducted a detailed performance study on Silo [15], a recent state-of-the-art main-memory database. Compared to Shore, the elimination of buffer manager (centralized page manager) significantly improves table operation (record access) performance, and we found that index operation is now the new bottleneck in Silo. In terms of cache performance, misses mostly happen at last-level cache for table operations, indicating poor locality among table records. The RCU (read-copy-write) region (to support concurrent access) of the underlying B-tree indices also causes a significant number of last-level write misses.

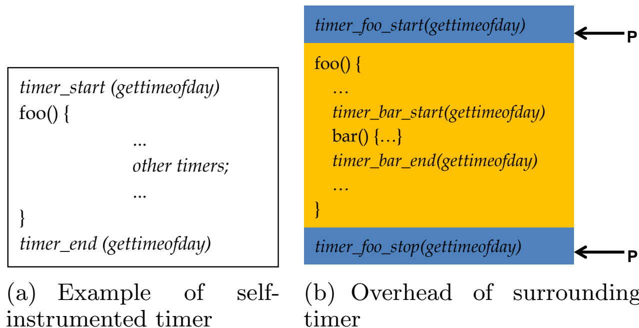
In addition, we examined the overhead of Silo's OCC protocol. We found that under normal workloads, OCC still imposes a significant overhead during commit time, especially when transaction is short and write intensive. As the workloads become skewer or there are more threads, OCC overhead starts to dominate mainly because the data read during transaction execution is more likely to be modified by other threads, causing transaction abort. We also report our findings as we vary the transaction types. For example, we found that the abort rate peaks at the point where there are 50% read transactions and 50% write transactions.

The remainder of this report is organized as follows. Section 2 gives an overview of our target system Silo. Section 3 introduces the measurement methodology as well as the benchmark we are using. We report our measurement result in section 4. Finally, we conclude our report in Section 5.

2. SILO OVERVIEW

Silo is a state-of-the-art main-memory relational database that is optimized for multi-core machines. Silo can achieve up to 700,000 transactions per second running standard TPC-C benchmark, and up to 15 million transactions per second running YCSB workload A benchmark using 32 cores [15].

Silo's underlying backbone is Masstree [12], a high-performance concurrent key-value store that also support range queries. In fact, tables in Silo are logical. They are implemented as collections of Masstrees (as index trees), with all the tuples (database records) hanging on the leaves of Masstrees [15]. The basic structure of Masstree is a concatenation of layers of B+-trees that conceptually form a trie [12]. In other words, the point-



(a) Example of self-instrumented timer

(b) Overhead of surrounding timer

Figure 1: Timer overhead analysis

ers in a leaf node may either point to a data record (if the key is already unique at that point), or to a lower-level B+-tree for further search on the next keyslice. The shared prefixes of keys, chopped into fixed-length (8 bytes by default) keyslices, are indexed by layers of the B+-trees, while the unique suffix for each key is stored separately in the leaf node.

Masstree achieves high-performance for 2 major reasons. First, its trie-like structure is optimized for key comparison [15]. Each layer of B+-trees is only responsible for indexing an 8-byte keyslice. That means keyslice comparison at each layer only involves integer comparison instead of string comparison (which is much slower). Second, the structure and fanout of the B+-tree nodes are carefully designed such that every node occupies almost exactly 4 cachelines (experiments show that 4 cahcelines is the optimal size in this case) [12]. Such cache-friendly design significantly not only reduces data cache misses, but also utilizes the prefetching hardware well.

Silo implements a variant of optimistic concurrency control (OCC) in its transaction layer on top of Masstree to ensure required serializability. As the transaction is running, it maintains a read-set and a write-set that identifies all the read records and modified records respectively, along with its transaction ID (functioning as a timestamp) [15]. The commit protocol is divided into 3 phases. In phase 1, the transaction tries to acquire write locks for every record in the write-set. Phase 2 does the read validation. If some record in the transaction's read-set no longer has the latest transaction ID (i.e. it has been modified by some other thread), the whole transaction will abort. And finally, in phase 3, the transaction commits its updates to the underlying Masstree and releases the write locks [15]. OCC is considered beneficial for scalability because it only has a very short period (at commit time) to write to shared memory, which reduces contention [11].

3. MEASUREMENT MECHANISM

To the best of our effort, we want to use existing profiling tools as much as we can to measure the performance of Silo, since our focus is to understand Silo's behavior but not to invent new measurement tools. However, through our measurement, we found that existing profiling tools often have limitations regarding either multithreading support or programming language support. To get a more accurate understanding of Silo, we will use our own timers to instrument the original code in some specific scenarios.

3.1 Valgrind

For most of our measurements, we use Valgrind [3] as the profiling tool. Valgrind is a popular profiler that runs every target program in a virtual machine based on just-in-time compilation. The program that runs on top of Valgrind will be first translated into an intermediate state, in which it can be analyzed using modular tools. Multiple tools have been created. We use Callgrind, a popular tool which is able to capture instruction fetch and cache performance. We also use the *–toggle-collect* option in Callgrind to easily collect interesting events regarding to one kind of transaction at a time.

Because of the virtual machine layer, the program running on Valgrind has a slightly different working environment than running on the host. Therefore, the program's behavior can also be different. These differences are usually acceptable to us except that Valgrind serializes the program so only one thread runs at a time [2]. This will cause a huge difference to Silo's behavior under contention between threads.

We have tried some other profiler tools to see if they have better multithread support. Gprof seems to be a good candidate. However, as far as we know, we cannot use gprof to collect information from only one class function in C++. We finally decided to use our own timers to study Silo's performance under contention.

3.2 Self-instrumented timers

Our own timers are based on the *gettimeofday* system call, which is consistent with the original measurement method in Silo's paper. Figure 1(a) explains how we instrument the original Silo code to measure the time spent on function *foo()*. Basically, we get the system time before and after calling the function we are interested in. The difference is than roughly the function execution time.

However, since a lot of the function we want to measure is quite fast, maybe even faster than a timer itself, we must remove the timer overhead in the measurement. There are two kinds of timer overhead we are considering.

First, timer overhead within the function under measurement. For example, in Figure 1(b), *bar()* is a subfunction inside of *foo()*. We add timers around both *foo()* and *bar()*, since both of them are of our interest. Therefore, the measurement of execution time of *foo()* has an overhead of about twice the *gettimeofday*

execution time.

Second, timer overhead around the function under measurement. Again, in Figure 1(b), when measuring the function execution time of *foo()*, the timer around it also has an overhead. Now let's calculate it. *gettimeofday* gets the system time at one specific point. Although we do not know which point it is, we know it is somewhere within the execution of the gettimeofday system call. Assume it's at point P, as in figure 1(b). We assume the relevant position of the time point is consistent each time we call it, so that the time point we get for *timer_foo_stop* will be somewhere near P'. It is then obvious that the surrounding timer overhead of measuring any function is the same as one *gettimeofday* execution time.

For both kinds of overhead mentioned above, we exclude them in our measurement. We wrote a small program to run *gettimeofday* for 1 billion times and calculated that running it once has an overhead of about 20ns in our system.

3.3 Workload

We use two standard benchmarks to study Silo's performance. The first one is TPC-C, a popular OLTP benchmark which tries to simulate real-world online transaction workloads. Among the five transaction types, *New Order* and *Payment* account for around ninty percent of transactions in a standard workload [8]. Therefore, we measure Silo's performance in such two transaction types.

Another benchmark we use is YCSB. It is a popular key value store benchmark, providing tunable parameters to specify the data access pattern. Therefore, this is the ideal benchmark for us to study Silo's performance under contention.

However, Silo's codebase does not provide an interface implementation to use the full YCSB features. It only implements a random distribution. In order to better understand Silo's behavior under contention, we implemented the Zipfian distribution to fit Silo's interface. Zipfian distribution has been regarded as a good way to characterize contention for database systems. [5]

4. PERFORMANCE STUDY

4.1 Experiment setup

All of our experiments run in an Dell Optiplex 9010 machine. It has four Intel Core i7 3.40 GHz cores with hyperthreading support. The memory size is 32 GB. The machine is running Ubuntu 12.04.

For section 4.2 and 4.3, we use Valgrind to profile the original Silo code. The codebase is nearly unchanged. However, since Valgrind cannot capture function information if a function is inline, we have specifically made several functions not inline to ease collection of relevant data. For these two sections, we use the YCSB benchmark with random distribution. For secion 4.4, we use our own timers and the Zipfian distribution.

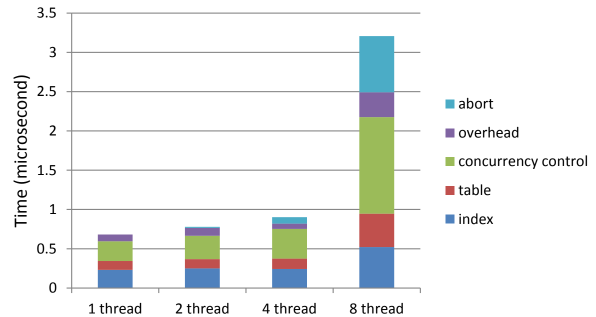


Figure 4: Performance Under Contension: Varying Number of Threads

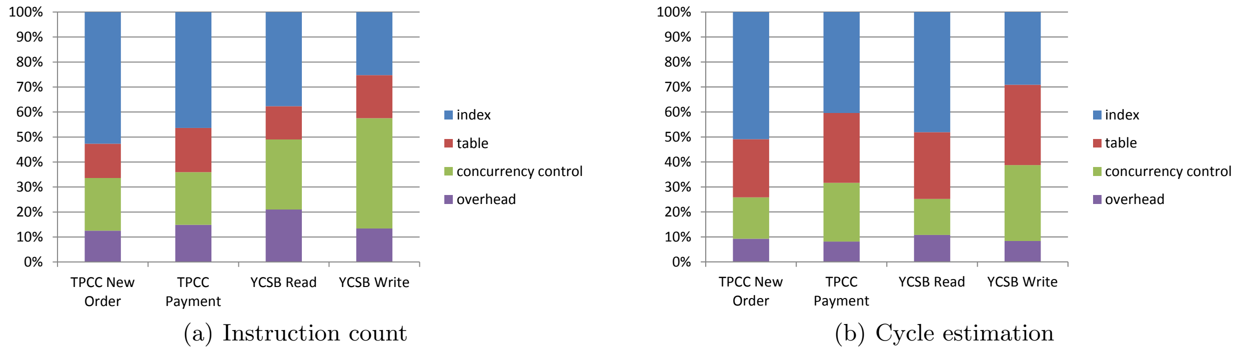
In all TPC-C workloads, we have set the number of warehouse to be 8. This gives us a database of 70MB in total size. For performance breakdown of YCSB workloads, we used eight hundred thousand keys, which gave us a 40MB database. When using Zipfian distribution, we set the database to be much smaller, with only two thousand keys. This is because we are loading the precomputed Zipfian distribution into memory since generating random key according to Zipfian distribution at run time is too much an overhead for a transaction. We set the pre-loaded Zipfian workloads small enough so that it won't introduce too much additional cache misses. However, since the whole database is much smaller, the cache performance for this set of experiments may not reflect its real-world characteristics.

4.2 Instruction count

Figure 2(a) shows the performance breakdown of Silo in terms of instruction count for four different transaction types, new_order and payment in TPC-C, as well as read and write in YCSB. We classify all Silo operations into four categories, index operation, table operation, operations related to concurrency control, and the benchmark overhead. Since this result is collected through Valgrind, it can be regarded as single thread performance breakdown of Silo, without any inter-thread contension.

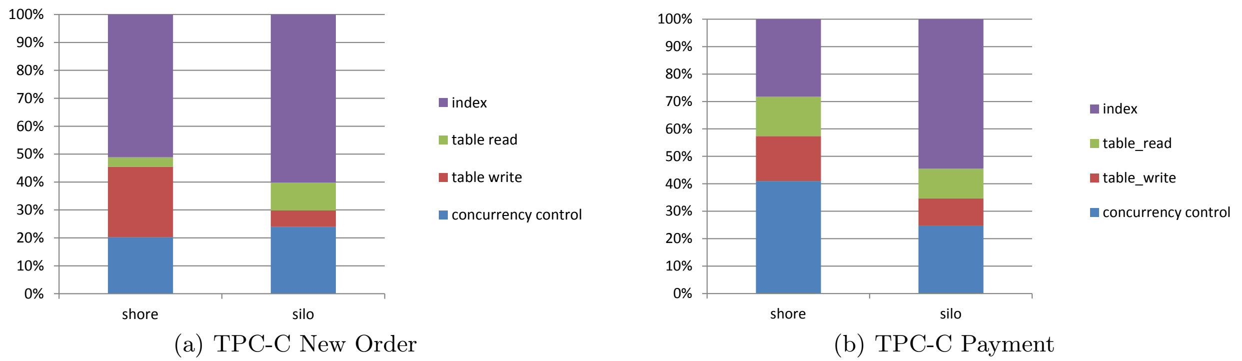
It is clear in the figure that index operations account for a significant amount of instruction counts for all the transaction types. Particularly, in the two TPC-C transactions, index operations can take more than half of the total instruction counts. Concurrency control overhead is not significant for TPC-C transactions, but rather large for YCSB transactions. This is because YCSB transactions are usually very short. In all four transactions, table operations are usually very fast, accounting for less than 20 percent of the instructions.

We compare our performance breakdown of Silo to that of Shore. Since Shore only reported results for



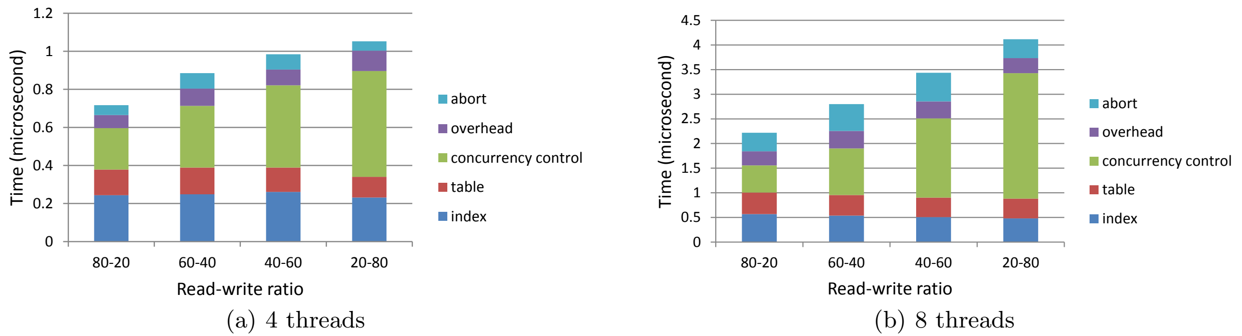
(a) Instruction count (b) Cycle estimation

Figure 2: Performance Breakdown of Silo



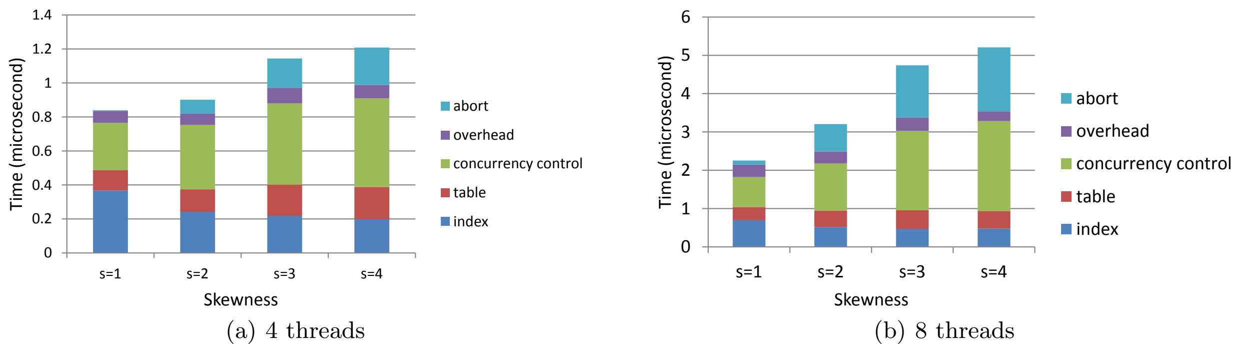
(a) TPC-C New Order (b) TPC-C Payment

Figure 3: Performance Breakdown Compared to Shore using Instruction Count



(a) 4 threads (b) 8 threads

Figure 5: Performance Under Contension: Varying Workloads (Skewness = 2)



(a) 4 threads (b) 8 threads

Figure 6: Performance Under Contension: Varying Skewness (Read/write ratio = 1 : 1)

TPC-C workload, we compare our results using the same two transactions. The comparison is shown in figure 3. In shore, index operation, table operation, and concurrency control take roughly the same amount of instruction count, but in Silo, it is clear that index operation is now a bottleneck. This is consistent with the past several years of development in major database systems. The removal of buffer manager has significantly reduced the time overhead of managing table data. OCC is also a better fit for main memory databases for concurrency control. However, the improvement of index operations is not much.

4.3 Cycle estimation and cache performance

Using the same set of workloads, we show the performance breakdown of Silo in term of cycle estimation. The cycle estimation used in Valgrind takes into account both instruction count and cache performance. Specifically, we set the parameters so that a L1 cache miss accounts for 25 cycles and a last-level cache miss (access of memory) accounts for 300 cycles. The result is shown in Figure 2(b), from which we can see that index operation is still a bottleneck.

Compared to Figure 2(a), we can see that table operation accounts for a larger portion in terms of cycles than in terms of instruction counts, indicating there are more cache misses in table operation. According to the Valgrind cache simulation, table operation has a 4% to 7% L1 data miss rate, and a 3% to 5% last-level data miss rate for TPC-C workloads. These numbers are siginificantly higher than other operations. We also noticed that for table operations, misses mostly happen at last-level cache, indicating that table records generally have poor locality. This is because tables in Silo are logical. They are implemented as collections of Masstrees (as index trees), with all the tuples (database records) hanging on the leaves of Masstrees [15].

Concurrency control operations also has a relatively high last-level cache miss rate. This is because Masstree uses an RCU (read-copy-write) region for versioned values to support concurrent access. Memory allocation of the RCU regions usually causes significant amount of last-level misses. On the other hand, index operations have a relatively lower last-level miss rate, usually no more than 1.5%.

4.4 OCC overhead under skewed workload

In this section, we use our own timers to study Silo's performance under contention. In addition to index, table, and concurrency control operations, we also look at how much time is wasted because of a transaction being aborted. Figure 4 shows how many time is spent on each transaction using different number of threads. In this experiment, the skewness of YCSB workload is set to 2, and half of the transactions are read, half are writes. As we can see from the figure, as there are more threads, there are more time wasted on aborted operations because it is more likely that the data read has been modified during transaction execution. The concurrency control overhead also becomes dominant, since there will be a larger collection of write sets to be checked for each transaction.

There is a surprisingly large performance differnce between 4 threads and 8 threads. We believe this is becasue the machine we run experiment on does not have eight real cores, so that eight threads cannot be run in parallel. The hyperthreading support can only siginificantly increase system's performance for multi-threaded programs when a single thread cannot fully leverage all the functional units in one core. This is probably not the case for the database workloads under our test.

We also test Silo's performance under contension by varying read-write ratio (Figure 5) and skewness (Figure 6). As expected, when there are more write transactions, or the workload is more skewed, there is more time spent on concurrency control operations. Another interesting observation is that abort rate is the highest when the number of read/write transactions in the workload is roughly the same. This is because aborts happen only when a transaction's read set has been modified by other thread. If read ratio is high, OCC works best; If write ratio is high, abort rate is relatively low because there are not enough reads to be invalidate. However, it is obvious in the figure that as the workload becomes write intensive, OCC commit overhead increases significantly. Based on this observation, we conclude that write lock acquiring (and contention), other than transaction abort, is the main killer of OCC under write intensive workloads.

5. CONCLUSION AND FUTURE WORK

In this work, we conducted a detailed performance study of a state-of-art main-memory database system, Silo. We used both existing profiling tools as well as our own instrumented timers to analyze Silo's performance under normal and skewed workloads. We found that for normal OLTP workloads, index operations are a clear bottleneck. As the number of threads, the skewness of workloads increases, the concurrency overhead becomes dominant, especially for shorter transactions. The number of transaction aborts peaks at the point where the number of read transactions is equavalent to that of write transactions.

We believe siginificant effort is needed in index optimization to further improve database performance. For example, instead of storing the whole key for each unique suffix, we can store only the hash of the key and check at read. This will reduce the total size of index tree and speed up index operations. To deal with the

high overhead of concurrency control under contension, it may be benifitial to dedicate one thread to schedule different transactions to different cores. By grouping transactions that access the same data and assigning them to one core, there can be a siginifcant reduction in concurrency control overhead and aborted transactions.

6. REFERENCES

[1] Shore Database. `http://research.cs.wisc.edu/shore/`.

[2] Valgrind Manual. `http://valgrind.org/docs/manual/manual-core.html`.

[3] Valgrind Wiki. `http://en.wikipedia.org/wiki/Valgrind`.

[4] VoltDB. `http://www.voltdb.com/`.

[5] Zipfian distribution. `http://en.wikipedia.org/wiki/Zipf\%27s_law`.

[6] Diaconu, C., Freedman, C., Ismert, E., et al. Hekaton: SQL server's memory-optimized OLTP engine. *Proceedings of the 2013 international conference on Management of data*, pages 1243–1254, June 2013.

[7] Forbes. $9.3 Billion Sales Recorded In Alibaba's 24-Hour Online Sale. http://www.forbes.com/sites/hengshao/2014/11/11/9-3-billion-sales-recorded-in-alibabas-24-hour-online-sale-beating-target-by-15/.

[8] Harizopoulos, S., Abadi, D. J., Madden, S., Stonebraker, M. OLTP through the looking glass, and what we found there. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992, June 2008.

[9] IBM. DB2. `http://www-01.ibm.com/software/data/db2/`.

[10] Kallman, R., Kimura, H., Natkins, J., et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[11] Kung, H. T., Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[12] Mao, Y., Kohler, E., Morris, R. T. Cache craftiness for fast multicore key-value storage. *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, April 2012.

[13] Mu, S., Cui, Y., Zhang, Y., Lloyd, W., Li, J. Extracting more concurrency from distributed transactions. *Proceedings of OSDI*, October 2014.

[14] TechCrunch. Alibaba Smashes Its Record On China's Singles' Day. http://techcrunch.com/2014/11/10/alibaba-makes-strong-start-to-singles-day-shopping-bonanza-with-2b-of-goods-sold-in-first-hour/.

[15] Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S. Speedy transactions in multicore in-memory databases. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, November 2013.

[16] Zheng, W., Tu, S., Kohler, E., Liskov, B. Fast databases with fast durability and recovery through multicore parallelism. *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 465–477, October 2014.