Infinite Resources for Optimistic Concurrency Control

Theo Jepsen
Università della Svizzera italiana

Leandro Pacheco de Sousa
Università della Svizzera italiana

Masoud Moshref
Barefoot Networks

Fernando Pedone
Università della Svizzera italiana

Robert Soulé
Università della Svizzera italiana
Barefoot Networks

ABSTRACT

Optimistic concurrency control (OCC) is inefficient for high-contention workloads. When concurrent transactions conflict, an OCC system wastes CPU resources verifying transactions, only to abort them. This paper presents a new system, called Network Optimistic Concurrency Control (NOCC), which reduces load on storage servers by identifying transactions that will abort as early as possible, and aborting them before they reach the store. NOCC leverages recent advances in network data plane programmability to speculatively execute transaction verification logic directly in network devices. NOCC examines network traffic to observe and log transaction requests. If NOCC suspects that a transaction is likely to be aborted at the store, it aborts the transaction early by re-writing the packet header, and routing the packets back to the client. For high-contention workloads, NOCC improves transaction throughput, and reduces server load.

CCS CONCEPTS

• **Networks** → **In-network processing**; • **Information systems** → **Distributed database transactions**;

KEYWORDS

Programmable switches; Optimistic concurrency control

NetCompute'18 '18, August 20, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5908-5/18/08…$15.00
https://doi.org/10.1145/3229591.3229597

ACM Reference Format:
Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. 2018. Infinite Resources for Optimistic Concurrency Control. In *NetCompute'18 '18: Morning Workshop on In-Network Computing, August 20, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229591.3229597

1 INTRODUCTION

Optimistic concurrency control (OCC) [21] is a key technique used by storage systems to ensure correctness in the presence of concurrent transactions. With optimistic concurrency control, a storage system speculatively executes a transaction without acquiring locks. Before committing a transaction, t, the storage system must verify that no other transaction has modified the data that has been read by t.

This *optimistic* approach stands in contrast to two alternative mechanisms for concurrency control: *blocking* and *immediate restart* [1]. Blocking and immediate restart are both pessimistic approaches based on locking objects before a transaction executes. The mechanisms differ in how they handle denied lock requests. In the first approach, the transaction blocks until locks are acquired. In the latter approach, the transaction is immediately aborted and must be retried.

There have been numerous studies comparing the performance of OCC to pessimistic concurrency control [2, 8, 8, 13, 30]. Many of these studies have contradictory results. Carey and Stonebraker [8] argue that blocking provides better performance than restarts. Tay [30] argues that restarts provide better performance than blocking. And Franaszek and Robinson [13] argue that optimistic methods are preferable to pessimistic approaches.

A landmark paper by Agrawal et al. [1] sheds some light on why these reports disagree. The authors identify three important assumptions on which the prior works differ: (i) the existence of infinite resources; (ii) whether or not restarted transactions are replaced with new independent transactions; and (iii) whether read operations use exclusive locks, or shared locks that may be upgraded to exclusive locks.

This paper focuses on optimistic concurrency control and the infinite resource assumption.

By "infinite resources", Agrawal et al. mean an infinite number of CPUs. Given an infinite number of CPUs, in the absence of contention, throughput should be a function of the number of concurrent transactions (since each CPU could process a separate transaction in parallel). Note that the likelihood of conflicts increases with the number of concurrent transactions. Agrawal et al. showed in simulation that, assuming the existence of infinite resources, OCC throughput continues to increase with the amount of concurrency since it never needs to acquire locks. In contrast, the throughput of blocking and immediate restart would both plateau.

However, in practice, with a bounded number of CPUs, OCC is inefficient under high contention. Its performance degrades, since OCC wastes CPU resources to verify transactions, only to abort them. In other words, the throughput of OCC is bounded by the number of non-conflicting parallel executions of transactions.

In this paper, we argue that recent advances in programmable network hardware [6, 33] allows us to develop an OCC system that behaves as if it had infinite resources. The key idea is a new technique called *speculative verification offload*, which executes verification logic in the network. We have implemented speculative verification offload in a system named Network Optimistic Concurrency Control (NOCC). NOCC identifies transactions that are likely to abort in a Top-of-Rack switch, and aborts them before they reach the store. Thus, transactions that reach the store rarely abort, avoiding wasted server CPU, resulting in an extremely efficient optimistic concurrency control.

Motivation. To demonstrate the above behavior, and motivate NOCC, we performed a simple experiment in which we increased contention, and measured the throughput of successful transactions. To increase contention, we increased the number of clients attempting to read and modify (increment) the same object in a key-value store. All the transactions passed through a switch that operated in one of two different modes of execution. In the first, the switch acted traditionally, and simply forwarded requests to the store. In the second configuration, which will be explained in Section 2, the switch executed NOCC logic to offload verification.

Figure 1 demonstrates that the performance of OCC degrades under high contention. With the traditional OCC store, as the number of clients increases, the store sends more aborts per transaction. In contrast, NOCC is much more efficient. Since the switch performs the validation of transactions, most conflicting transactions are aborted in the network, resulting in higher throughput.

Contributions. We have implemented a prototype of NOCC using the P4 language [5], and evaluated it using Barefoot

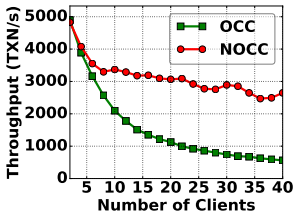


Figure 1: Throughput for incrementing a single counter as contention increases.

Network's Tofino ASIC [6]. Our experiments include both a set of micro-benchmarks that explore the parameter space, and an implementation of TPC-C [9] to emulate a real-world workload. Our evaluation shows that under high-contention workloads, NOCC significantly increases transaction throughput and reduces server load. Under low-contention workloads, NOCC adds no additional overhead.

Overall, this paper makes the following contributions:

- It presents a novel algorithm for improving the performance of optimistic concurrency control by offloading verification logic to the network.
- It describes an implementation of the speculative verification offload technique that builds on emerging technological trends in programmable data planes.
- It explores the parameter space for network-based transaction verification, and demonstrates significant performance improvements for high-contention workloads.

The rest of this paper is organized as follows. We first describe the design of NOCC (§2) and the details of its implementation (§3). We then present a thorough evaluation (§4). Finally, we discuss related work (§5), and conclude (§6).

2 DESIGN

Before presenting the design of NOCC, we briefly provide important definitions and describe the system model (i.e., key aspects of the system and environment).

Definitions and System Model

We consider a distributed system composed of *client* processes and a *store*. Processes communicate through message passing and do not have access to a shared memory. The system is asynchronous; We do not assume any bound on messages delays and on relative process speeds. Processes are subject to crash failures and do not behave maliciously (e.g., no Byzantine failures).

The store contains a set $D = \{x_1, x_2, ...\}$ of data items. Each data item x is a tuple $\langle k, v \rangle$, where k is a key and v a value. We assume that the store exposes an interface with two operations: *read(k)* returns the value of a given k, and *write(k,v)* sets the value of key k to value v. We refer to those transactions that contain only read operations as *read*

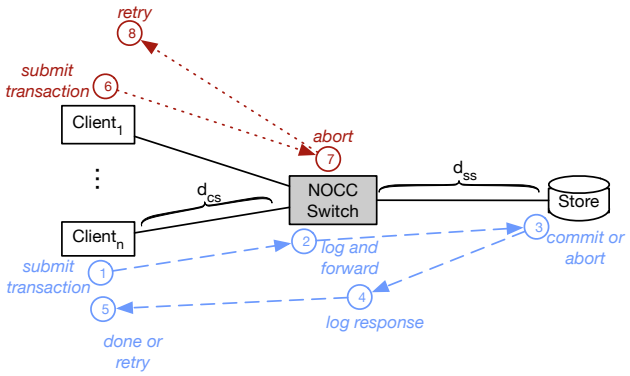


Figure 2: Overview of NOCC deployment.

transactions. Transactions that contain at least one write operation are called *write transactions*.

We assume that clients execute transactions locally and then submit the transaction to the store to be committed. When executing a transaction, the client may read values from its own local cache. Write operations are buffered until commit time.

The isolation property that the system provides is *one-copy serializability*: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [4]. To ensure consistency, the store implements optimistic concurrency control. All *read transactions* are served directly by the store. To commit a *write transaction*, the client submits its buffered writes together with all values that it has read. The store only commits a transaction if all values in the submitted transaction are still current. As a mechanism for implementing this check, the system uses a *compare(k,v)* operation, which asserts that the value of k is v (i.e., the value has not changed since the client's last read. In the event of an abort, the server returns *corrections* with the up-to-date values that caused the compares to fail. This allows the client to immediately re-execute the transaction.

System Overview

Figure 2 shows a basic overview of NOCC. Transaction requests pass through a NOCC switch, which either forwards the request on to the store, or aborts the transaction and responds to the client directly.

The blue, dashed-line shows the forwarding case: (1) the client submits the transaction; (2) the switch logs the transaction in its local cache, and forwards it to the store; (3) the store decides to commit or abort the transaction and responds with the decision; (4) the switch logs the result of the execution, and forwards the response to the client. (5) the transaction either completes or the client must re-try.

The red, dotted-line shows the abort case: (7) the switch examines the transaction message. If the switch sees that the transaction is likely to abort based on some previously seen transaction, the switch preemptively aborts the request; (8) Upon receiving the abort message, which contains corrections, the client can re-submit the transaction.

For a transaction that would have aborted at the store, there are two advantages of the *speculative verification offload* approach. First, the store does not waste resources on verifying the transaction, reducing load on the store. Second, the message avoids traveling the distance from the switch to the store, d_{ss}, twice.

Data Store. A transaction request message contains three possibly empty lists of operations: *compares*, *reads*, and *writes*. The store first checks the compares for stale values. If any compare fails, the store aborts the transaction. As part of the abort response, the store includes a list of correct values, with the updated values for comparisons that caused the transaction to fail. Otherwise, the store updates the values of its data items with the values from the writes. Then the store responds to the client with all the values that were updated, along with the values that the transaction may read.

Speculative Verification Offload. The NOCC switch logs requests and responses from several clients in order to determine if a subsequent transaction is likely to abort. NOCC adopts an aggressive strategy for aborting transactions. It proactively updates its cache with the latest value after the switch has seen a transaction request (step 2 in Figure 2). Note that this cache is only used to make decisions about aborting, and does not serve read requests.

We refer to this as a *speculative verification offload* strategy. It is speculative because the switch assumes that any transaction request that it has seen and conforms to its cached values is likely to be committed. As a result, it can make decisions about aborting subsequent transactions sooner. However, this approach may abort transactions that would not have been aborted by the store, which we discuss in Section 4.

The logic for speculative verification offload is as follows. The switch has logic for processing both transaction requests and their responses. When the switch receives a transaction from the client, it checks the compares. If any compare operation references a key that is not in the cache, then the switch cannot reason about the validity of the transaction, so it forwards the request to the store. If the compare references a key that is in the cache, then the switch compares the value in the packet with that in the cache. If the values differ, the value in the cache is added to a per-transaction set of corrections. After processing the compares, if there is at least one correction (i.e. there was a comparison that failed), the switch immediately sends an abort response to the client with the set of corrections. When the client receives the abort response, it uses the values in the corrections to recompute and resubmit the transaction immediately, avoiding an extra

round trip of requesting the latest value from the store. There are two benefits: the client can retry the aborted transaction right away (lower latency); and the there is less chance that the value will have changed again since receiving the abort (which would be more likely if the client had to re-request the value).

If no comparisons fail, then the switch checks if the request contains write operations. If there are write operations, the switch updates its cache with the new values. Then, it forwards the transaction to the store for processing.

An abort message from the store contains a non-empty list of the corrections. The corrections contain the updated values that caused the transaction to fail. When the switch receives an abort message, it updates its cache with the correct values. If the switch did not update its cache on aborts, it would incorrectly abort subsequent transactions.

We note that although a NOCC switch needs to maintain states in its local cache, the size of the cache does not need to be too large to be effective. It is sufficient to reserve enough space for "hot" data items. The amount of space available to the cache will depend on the target platform for deployment. If the size is restricted, NOCC could use a cache eviction policy to make space available for new items.

Detecting Stale Values. In a typical transactional storage system, data items would include a version number that the store would use to determine if a transaction should be aborted due to a stale value. However, with the speculative verification offload strategy, the switch must update its local cache of data items before the store could assign a version number. Therefore, NOCC cannot use version numbers to check for stale values. Instead of comparing version numbers, NOCC compares the actual values of data items. Furthermore, by storing the actual values on the switch, they can be included in switch-generated abort messages, which enables clients to retry transactions immediately with the latest values. This obviates the extra round trip of the client requesting the latest value from the store, reducing transaction retry latency and load on the store.

Expected Deployment. We expect that NOCC would be deployed in a Top-of-Rack switch that inspects all traffic in a rack of storage servers. However, if NOCC were deployed in a way that it did not interpose on all traffic to a store (e.g., clients connected to different switches update data at the store), NOCC does not violate correctness. Clients and switches will learn of new values after an abort message from the store. For example, if client_1 writes a value v_1 for key k_1, then switch_1 will record v_1 in its cache. However, if client_{n+1} had previously written a value v_1' for key k_1, the request from client_1 will pass through switch_1, but will be aborted by the store. The client and switch_1 will learn of the new value v_1' in the abort response from the store. They will both then update their local caches, and the client can re-submit the transaction with the latest value.

Correctness. The store ensures one-copy serializability. The serialization order is defined by the arrival order of transactions at the store. A transaction only commits if all the reads it performed during execution are still up to date at the time the transaction is received by the store.

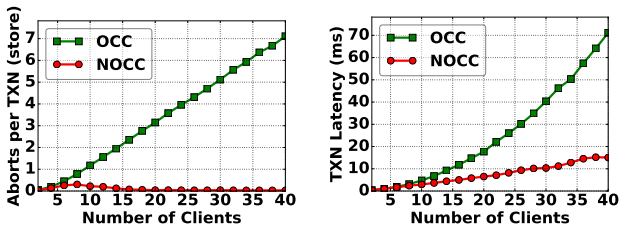
The correctness of the switch logic follows from the fact that (a) the switch does not commit any transaction, although it may abort transactions, and (b) the switch forwards non-aborted transactions to the store without changing their operations. The switch may abort transactions that would not have been aborted by the store; this does not compromise correctness, but has a performance penalty which is outweighed by the benefits of correct aborts.

3 IMPLEMENTATION

We have implemented NOCC as a P4 program that runs on a Barefoot Tofino ASIC [31]. We have also implemented a client program and OCC transactional storage system in Python. NOCC uses a custom *transaction* header encapsulated in a UDP packet, followed by a sequence of fixed-width *operation* headers.

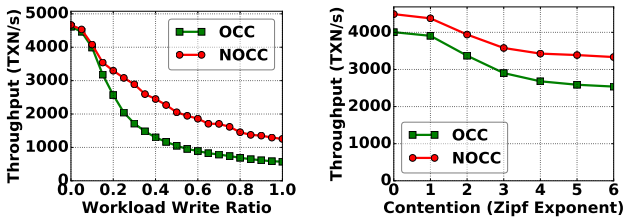
The P4 program contains parsers and tables for standard L2 forwarding, as well as processing logic for transaction packets, which is divided into two phases. In the first phase, the program iterates over the operations, checking that the compares are valid (i.e., the value in the packet matches the value in the cache). In the second phase, the program iterates over the operations a second time, either: updating the cache if the transaction is valid; or, updating the invalid values in the packet and returning it to the client as an abort.

The cached values on the switch are stored in registers which allow up to 32 bits to be read or written in a single pipeline stage [6]. This presented us with two challenges: caching large values and accessing them multiple times for the same packet (e.g., for the iterations). To store larger values, we split the value across multiple registers stored in different stages. To iterate over the operations in the packet, we use recirculation; after each iteration, the packet is recirculated through the pipeline, storing the state of the computation (including iteration index) in packet metadata. Although we could recirculate an arbitrary number of times, the number of operations per transaction we support is bounded by how deep the packet can be parsed, which in turn, is bound by the size of the packet header vector [6]. Although recirculation reduces the throughput of the switch, it still provides better performance than a software implementation.

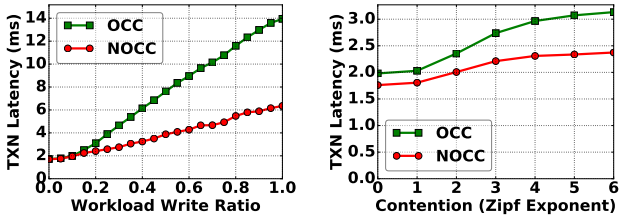


(a) aborts per txn vs. #clients (b) latency vs. #clients

Figure 3: NOCC has low aborts and latency at store.



(a) throughput vs. write ratio (b) throughput vs. contention



(c) latency vs. write ratio (d) latency vs. contention

Figure 4: NOCC improves throughput and latency as write ratio and contention changes.

4 EVALUATION

In this section, we describe two sets of experiments that evaluate NOCC. The first set of experiments are microbenchmarks that explore how NOCC impacts the performance for executing transactions on a key-value store under changing operational conditions: number of clients, write ratio, and workload contention (skew).

In the second set of experiments, we explore real-world inspired workloads, by using the TPC-C [9] benchmark with the parameters adjusted to increase contention. Overall, the results demonstrate that NOCC improves throughput and reduces latency for workloads with high contention.

Experimental setup. We used two machines, each with 12 cores (dual-socket Intel Xeon E5-2603 CPUs @ 1.6GHz), 16GB of 1600MHz DDR4 memory, and Intel 82599ES 10Gb Ethernet Controllers connected to a Barefoot Tofino switch. All the clients were collocated on one machine, while the store server ran alone on the other machine.

Microbenchmarks

NOCC has higher throughput as the write ratio increases. Each client in the benchmarks issues a mix of *read* and *write* transactions on the same key. The write ratio dictates the percent of the total number of transactions that are writes. Figures 4a and 4c show throughput and latency for 8 parallel clients, as the write ratio changes. These figures clearly demonstrate the effect of write operations, which limit the overall performance of the system. As the write ratio grows, OCC's throughput becomes limited due to the high cost of aborting writes. NOCC's throughput, on the other hand, degrades slowly, since requests are aborted at the switch and clients can optimistically retry transactions sooner. At 0.2 write ratio, NOCC's throughput is already 1.3x that of OCC, reaching 2.2x that of OCC at 1.0 write ratio.

NOCC does not add overhead for reads. When all transactions are reads, a NOCC switch does not perform verification logic. We can see in Figures 4a and 4c that when the write ratio is 0, NOCC has no overhead compared to OCC.

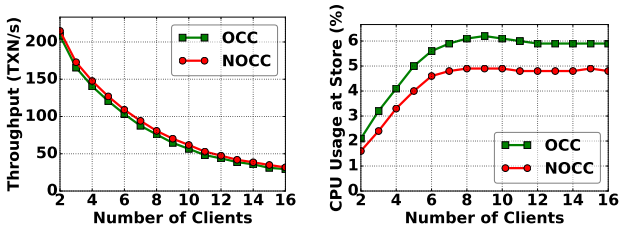
NOCC scales with the number clients. Figure 1 reports throughput when the write ratio is fixed at 0.2. NOCC provides a higher transaction rate, past the saturation point of OCC, reaching 4.6x the throughput at 40 clients. Using NOCC, transactions can abort early and optimistically be retried with the latest values, *before the previous conflicting transaction commits.* Offloading verification to the switch reduces the number of aborts the store has to send (Figure 3a), freeing the store's resources to commit valid transactions. Because of this, the transaction latency scales linearly with the load, as Figure 3b shows.

Skewed workloads benefit from NOCC. This experiment characterizes the effect of contention on the performance of NOCC. Clients submit transactions that can access one of 10 keys, and the popularity of each key is dictated by a Zipf distribution. Figure 4b shows how increasing the Zipf exponent affects throughput. With the Zipf exponent at 0, all keys are accessed with the same probability, and contention increases for larger exponents. As contention increases, the number of aborts grows, limiting the performance of write transactions for OCC. NOCC, on the other hand, by optimistically aborting and retrying transactions closer to the client, is less affected by the increase in contention.

TPC-C

The TPC-C [9] benchmark models an online transaction processing (OLTP) workload for a fictional wholesale supplier that maintains warehouses for different sales districts.

We note that TPC-C is not a good benchmark for evaluating NOCC, since the benchmark is intentionally designed to avoid queries that result in high-contention. Nevertheless,



(a) #clients vs. throughput **(b) #clients vs. CPU usage**

Figure 5: TPC-C Payment transaction

TPC-C is a widely accepted standard for evaluating transaction processing systems. We therefore focus on the TPC-C payment transactions, which has contentious dependencies.

To further increase contention, we modified the parameters of the benchmark. The TPC-C specification states that the benchmark should be run with a set of specific parameter values: 1 warehouse, 10 districts, 3000 customers and 100,000 items. For our evaluation, we used the following settings: 1 warehouse, 2 districts, 10 customers and 10 items.

As a TPC-C driver, we modified an open source Python implementation from Pavlo et al. [26]. Since the store does not have indexes, we modified the transaction executor to only perform exact selects. To represent the TPC-C database in our store, we map each record to a key-value pair. The client driver keeps a copy of all records it has read or written, essentially mirroring the store with a local cache. This eliminates the unnecessary latency of issuing read requests for records that have not changed. Consistency is guaranteed by validating the transaction before committing, ensuring the read values (possibly from the local cache) are up-to-date.

Figure 5a shows the throughput for the Payment transaction, and figure 5b shows the CPU usage at the store, for increasing contention. In Figure 5b, we can see that NOCC reduces the load on the store (by up to 22% compared to OCC), while maintaining slightly higher throughput.

Discussion. After seeing the microbenchmark results, we expected TPC-C to maintain higher throughput with NOCC. However, the performance was limited by incorrect speculative decisions. The switch aborts transactions that would not have been aborted by the store. Nevertheless, NOCC performed better than the OCC baseline, suggesting that NOCC can speed-up transaction processing for high-contention OLTP workloads.

5 RELATED WORK

NOCC is superficially similar to Eris [23], in the sense that they both use programmable switches to accelerate transaction processing. However, they have very different execution models. The Eris model is based on prior work on independent transactions [10, 29], in which transactions are ordered first, and then executed. In contrast, with the NOCC model, clients pre-execute transactions locally, and then submit the result for validation (i.e., ordering).

Proxies and caches. The idea of using a proxy to extend distributed services is a well-established idea [28] that has been widely adopted [3, 7, 19, 20]. Proxies are often used to scale services by caching copies of data closer to clients, such as with content distribution networks (CDNs) [14, 25, 32]. CDNs typically are used for static content, although there are examples of proxies used for dynamic content [15]. Prior work has also explored the possibility of leveraging the network to route requests dynamically to proxies to service requests [32]. Notably, SwitchKV [24] uses OpenFlow-enabled switches to dynamically route read requests to proxy caches. NetCache [18] provides a P4-based implementation of a key-value store to cache hot-data items for highly skewed read workloads. NOCC differs from this work in that it is not a cache, per se. It keeps copies of transaction requests, but it does not service client read requests. Rather, it uses copies of previous requests to make informed decisions about when to abort transactions early, with the goal of reducing latency for write-heavy workloads.

Network Computing. Several recent projects have explored moving application logic into programmable network devices. Dang et al. [12] proposed the idea of moving consensus logic in to network devices. Paxos Made Switch-y [11] describes an implementation of Paxos in P4. István et al. [16] implement Zookeeper's atomic broadcast on an FPGA. Speculative Paxos [27] and NoPaxos [24] use programmable hardware to increase the likelihood of in-order delivery, and leverage that assumption to optimize consensus à la Fast Paxos [22]. NetChain [17] provides a network implementation of a coordination service.

6 CONCLUSION

NOCC moves transaction processing logic into the network, using a custom packet header and programmable switches to identify and abort doomed transactions as early as possible. For write-intensive, high-contention workloads, NOCC reduces load on the store and increases system throughput.

In the future, we envision combining NOCC with complimentary techniques for read-heavy workloads, e.g., using a cache to service read requests [18, 24].

Overall, concurrency control is a key component of storage systems, and NOCC demonstrates how tighter integration with the network can lead to significant improvements in performance.

Acknowledgments. We would like to thank Chang Kim for his support. This work was supported in part by SNF grant #200021_166132 and the Hasler Foundation project #16037.

REFERENCES

[1] Agrawal, R., Carey, M. J., and Livny, M. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst. 12*, 4 (Nov. 1987), 609–654.

[2] Agrawal, R., and Dewitt, D. J. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 529–564.

[3] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. Serverless Network File Systems. *ACM Transactions on Computer Systems (TOCS) 14*, 1 (Feb. 1996), 41–79.

[4] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[5] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR) 44* (July 2014), 87–95.

[6] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (Aug. 2013), pp. 99–110.

[7] Brewer, E. A., Katz, R. H., Amir, E., Balakrishnan, H., Chawathe, Y., Fox, A., Gribble, S. D., Hodes, T., Nguyen, G., Padmanabhan, V. N., Stemm, M., Seshan, S., and Henderson, T. A Network Architecture for Heterogeneous Mobile Computing. Tech. rep., University of California at Berkeley, 1998.

[8] Carey, M. J., and Stonebraker, M. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1984), VLDB '84, Morgan Kaufmann Publishers Inc., pp. 107–118.

[9] Council, T. TPC-C Benchmark, revision 5.11, 2010.

[10] Cowling, J., and Liskov, B. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (ATC)* (2012).

[11] Dang, H. T., Canini, M., Pedone, F., and Soulé, R. Paxos Made Switch-y. *SIGCOMM Computer Communication Review (CCR) 44* (Apr. 2016), 87–95.

[12] Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM Symposium on SDN Research (SOSR)* (June 2015), pp. 59–73.

[13] Franaszek, P., and Robinson, J. T. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst. 10*, 1 (Mar. 1985), 1–28.

[14] Freedman, M. J., Freudenthal, E., and Mazières, D. Democratizing content publication with coral. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2004), pp. 18–18.

[15] Grimm, R., Lichtman, G., Michalakis, N., Elliston, A., Kravetz, A., Miller, J., and Raza, S. Na Kika: Secure Service Execution and Composition in an Open Edge-Side Computing Network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (San Jose, California, May 2006), pp. 169–182.

[16] István, Z., Sidler, D., Alonso, G., and Vukolić, M. Consensus in a Box: Inexpensive Coordination in Hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2016), pp. 103–115.

[17] Jin, X., Li, X., Zhang, H., Foster, N., Lee, J., Soulé, R., Kim, C., and Stoica, I. Netchain: Scale-free sub-rtt coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2018).

[18] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In *ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017), ACM, pp. 121–136.

[19] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., Gifford, D. K., and Kaashoek, M. F. Rover: A Toolkit for Mobile Information Access. In *ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1995), pp. 156–171.

[20] Knutsson, B., Lu, H., Mogul, J., and Hopkins, B. Architecture and Performance of Server-Directed Transcoding. *ACM Transactions on Internet Technology (TOIT) 3*, 4 (Nov. 2003), 392–424.

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

[22] Lamport, L. Fast Paxos. *Distributed Computing 19* (Oct. 2006), 79–103.

[23] Li, J., Michael, E., and Ports, D. R. K. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017), ACM, pp. 104–120.

[24] Li, X., Sethi, R., Kaminsky, M., Andersen, D. G., and Freedman, M. J. Be fast, cheap and in control with switchkv. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2016), pp. 31–44.

[25] Nottingham, M., and Liu, X. Edge Architecture Specification, 2001. http://www.esi.org/architecture_spec_1-0.html.

[26] Pavlo, A. Python TPC-C. https://github.com/apavlo/py-tpcc, 2017.

[27] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2015), pp. 43–57.

[28] Shapiro, M. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *6th IEEE International Conference on Distributed Computing Systems (ICDCS)* (May 1986), pp. 198–204.

[29] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The end of an architectural era: (it's time for a complete rewrite). In *33th International Conference on Very Large Data Bases* (2007), pp. 1150–1160.

[30] Tay, Y. C., Goodman, N., and Suri, R. Locking performance in centralized databases. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 415–462.

[31] Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[32] Wang, L., Pai, V., and Peterson, L. The Effectiveness of Request Redirection on CDN Robustness. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2002), pp. 345–360.

[33] XPliant Ethernet Switch Product Family. www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.