NetStore: Leveraging Network Optimizations to Improve Distributed Transaction Processing Performance

Xu Cui
University of Waterloo
xcui@uwaterloo.ca

Michael Mior
University of Waterloo
mmior@uwaterloo.ca

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
kdaudjee@uwaterloo.ca

Sajjad Rizvi
University of Waterloo
sm3rizvi@uwaterloo.ca

Abstract

A tremendous amount of data is processed daily in the cloud, which inevitably introduces large amounts of data transfer within a datacenter network. Numerous studies have reported that the network is often the performance bottleneck for cloud applications. While flow scheduling techniques have been proposed to mitigate this performance problem, distributed network-aware applications have not received much attention. The advent of software-defined networking (SDN) enables leveraging flow scheduling to make cloud applications network-aware, opening up the potential to improve the performance of these applications.

In this paper, we propose co-designing network and cloud applications to optimize performance in the presence of datacenter network congestion. In particular, we built a network-aware distributed database framework which uses network state information to improve performance. We present two techniques to reduce transaction completion time for cloud database applications. First, we apply a novel load balancing algorithm at the network layer to support intelligent network route selection. Second, we introduce a network-aware caching algorithm to retrieve fresh data replicas from links in the network with low congestion. Our experimental results show that our techniques can significantly reduce average transaction completion times compared to an ECMP-based baseline.

CCS Concepts • **Information systems → Distributed database transactions**; **Distributed storage**;

Active'17, December 11–15, 2017, Las Vegas, NV, USA

ACM ISBN 978-1-4503-5167-6/17/12…$15.00
https://doi.org/10.1145/3155889.3155893

Keywords distributed transaction processing, software-defined networking, network-aware database

ACM Reference Format:
Xu Cui, Michael Mior, Bernard Wong, Khuzaima Daudjee, and Sajjad Rizvi. 2017. NetStore: Leveraging Network Optimizations to Improve Distributed Transaction Processing Performance. In *Proceedings of Active'17: International Workshop on Active Middleware on Modern Hardware (Active'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3155889.3155893

1 Introduction

With the increasing proliferation of big data, companies need to process terabytes of new data on a daily basis [15]. This processing of big data usually requires data transfer in the network. Since big data applications are usually deployed and run in the cloud, a high volume of data transfers can cause the datacenter network to become a performance bottleneck. In addition to big data applications, there are many other applications deployed in the shared infrastructure inside the cloud. Many of these applications are response time sensitive, and numerous studies have shown that delays of milliseconds can have a significant impact on business revenue [21].

However, due to the shared nature of data centers, applications neither have a global view nor control of resources when the datacenter network is congested, making it difficult to deliver good performance. With the network being a key shared resource, performance optimizations at the network layer are difficult to design, apply and control for distributed databases hosted within data centers. In this paper, we propose network-level optimizations to improve the performance of applications in a shared environment. In particular, we present a performant prototype system called NetStore by co-designing a distributed database framework with network optimizations in a shared cloud environment.

1.1 Problem Overview

Distributed storage systems often shard data across multiple servers to distribute load and increase storage capacity. Application programmers can perform optimizations based on data access patterns to achieve performance guarantees.

However, this is problematic for cloud environments because cloud networks exhibit on/off behaviours [7]. Network traffic spikes from time to time, which may cause network links to be saturated and add significant queuing delays. Cloud providers may overcome this bottleneck by over-provisioning network bandwidth in the system at the cost of wasted resources when the network is idle.

When part of the network is congested, our goal is to minimize the transaction completion times without sacrificing throughput. This is complicated by bandwidth constraints resulting from background traffic. Moreover, since we expect transactions to complete within a few milliseconds, effective optimization decisions must be made efficiently. For instance, for transactions with completion on the order of 10s to 100s of milliseconds, existing dynamic flow scheduling solutions such as Hedera [3] may take 10s of milliseconds to set up. Our target for optimization is a transactional key-value storage system.

1.2 Contributions

Our techniques leverage network-awareness to provide high performance transaction processing even in the event of high network utilization. While transaction traffic is unlikely to saturate the network, the presence of significant background traffic can cause poor response times. By co-designing the database framework and the network manager together, NetStore is able to make intelligent routing decisions to avoid congested paths and ensuing queuing delays. We introduce two complementary algorithms that use this information to route distributed transactions around network hotspots. Another advantage of this co-design is that we do not need to generate extra network traffic for our algorithms because the optimization decisions are piggybacked onto lock messages for transactions. Furthermore, we require no cooperation with other tenants on the network so our techniques can be deployed incrementally.

The main contributions of this paper are as follows:

- *Least Bottlenecked Path* (LBP), a novel load balancing algorithm that selects the least congested path among all the shortest paths between a pair of hosts for each network flow.
- Network-Aware Caching (NAC), a network-aware caching technique that uses knowledge of network traffic to cache data away from congested segments of the network
- NetStore, a prototype system which makes use of LBP and NAC to implement a low-latency transactional key-value store in the presence of unmanaged background traffic

2 System Architecture

We explore optimization opportunities in the context of a key-value storage system that processes transactions issued by clients in a data center network. We model a data center network using one of the most common topologies, the multi-rooted tree as shown in Figure 1[1]. A pod in the network consists of a number of interconnected servers and switches. In Figure 1, there are two pods each consisting of two aggregation switches, four top-of-rack (ToR) switches, or edge switches, and eight servers. Each ToR switch is connected to each aggregation switch within its pod and different pods are connected using links between core and aggregation switches. Cross-pod communications have to traverse the core link layer and cross-rack communications must traverse the aggregation layer. Typical data center network architectures will have oversubscription[2] at each layer of the network. For example, in our experiments, we use a network topology such that the core to aggregation links have an oversubscription factor of 2 and the aggregation to edge links have an oversubscription factor of 4. This gives the network a total oversubscription factor of 8:1 from core switches to servers.

Our transaction model provides strong consistency using 2PC and a centralized lock server with data stored in memory at each server. Each operation within a transaction is required to acquire a lock before execution. Combining strict two-phase-locking (2PL) with two-phase commit (2PC) provides strict serializability [9]. Details on the architecture and components of our prototype are described in Sections 2.1 and 2.2. Finally, we present our distributed data storage server in Section 2.3.

2.1 System Architecture Overview

Our sample application has two major components: data servers, forming a distributed transaction processing system, and a controller. The logically centralized controller not only acts as a lock server, but it also serves as a network manager. This network manager is responsible for setting up network routes among sources and destinations, as well as monitoring network conditions and making intelligent scheduling decisions based on the current network state. This design also allows our system to piggyback many control messages onto lock related messages.

As shown in Figure 2, client programs can interact with NetStore using a transactional API. Clients can aggregate multiple read or write operations and issue all the operations as a single atomic transaction. A client sends its transactions to the nearest data server in the network topology. The data server receiving the transaction will act as the coordinator. The coordinator requests the necessary locks via the controller, performs each operation, and sends the final result back to the client. In addition to lock management, the

[1]Note that the least bottlenecked path algorithm is designed to work with any datacenter topology where multiple paths between pairs of servers exist. Network-aware caching can work with any topology.

[2]Oversubscription [4] is a situation in which the total egress bandwidth is lower than the ingress bandwidth in a network switch or layer.

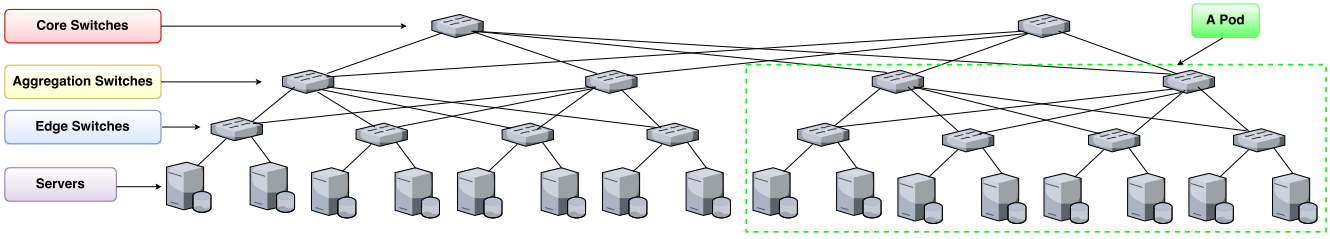

Figure 1. A Multi-rooted Tree Topology

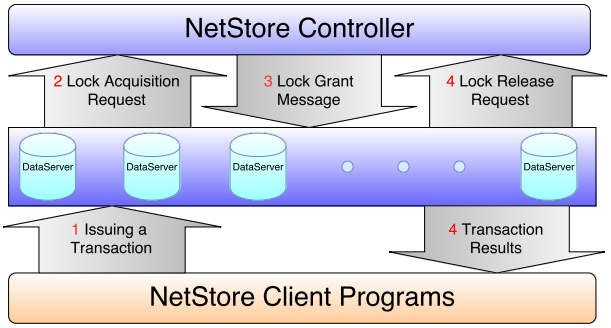

Figure 2. Prototype System Architecture

controller also determines which network path each operation should take. Furthermore, the controller maintains local metadata consisting of the locations and versions of cached data. This assists the controller in making network-aware caching decisions. Both the path information and caching decisions are piggybacked onto the controller's messages to coordinators. Upon receiving a response from a lock acquisition request from the controller, the coordinator will then perform the operations by interacting with its local storage or with a remote data server.

2.2 Controller

The controller performs three crucial tasks. First, the controller implements a read/write lock server with all operations in a transaction classified as read or write. The controller keeps updated information about lock acquisitions. To avoid deadlocks, the controller establishes a lock ordering based on transaction arrival times. With a two-phase-locking (2PL) implementation at the data server, this guarantees that no cycles can exist in the wait graph, avoiding deadlock.

Second, the controller also serves as a network topology manager. When the controller starts, it reads topology information such as link bandwidth from configuration files. Moreover, the controller discovers all of the switches and links in the topology through the link-discovery logic provided by the Floodlight SDN controller [2]. The controller aggregates all this information to set up path configurations between each pair of servers. In particular, if there are multiple paths between a pair of servers, switch configuration is required to distinguish each unique path.

To configure the switches, the controller establishes rules such that, in addition to the source and destination pair, the DSCP [1] bits in the TCP header allow the sending server to select a unique path to its destination. Furthermore, the controller stores metadata, with respect to each network link, to track the amount of data each flow transfers on each link at any given time. The overhead for these updates is minimal because they are performed only when the controller grants or revokes locks in response to lock acquisition or release requests. While it is possible to use the Floodlight SDN controller to estimate the amount of data transfer for background flows outside NetStore, we created our own background flow servers which send flow information to our controller before and after they send a flow into the network for simplicity.

Third, the controller maintains a local metadata cache of the real data entries maintained by the data servers. In this metadata cache, the controller stores keys, version numbers, and server locations for actual cache entries. This extra caching layer not only allows the controller to provide transactional consistency to NetStore without generating extra cache invalidation messages, but also to allow the controller to determine the most suitable server to read the cache replica from based on the network information.

With the ability to monitor network and the flow information, the controller is able to make intelligent path selections and caching decisions which greatly improve performance. The details are discussed in Sections 3 and 4.

2.3 Data Servers

The distributed data servers are in memory key-value stores. Each data server is responsible for distributed communication with other data servers as well as the controller. On startup, each server establishes TCP connections with other data servers for each unique path in the network. For each remote operation, while the data server is required to use the path selected by the controller, the data server is oblivious to the network topology. Servers only need to modify the DSCP value in the TCP header based on information sent by the controller. This design significantly reduces the complexity of the data server and leaves network logic at the controller.

Figure 3 depicts the lifecycle of a transaction. So far, we have discussed every stage in the transaction lifecycle except the two red boxes in this figure. In the next section, we

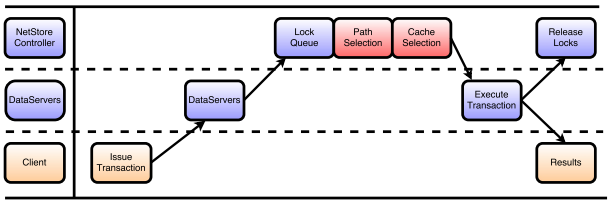

Figure 3. Transaction Lifecycle

examine how we achieve better performance by presenting the design details for our controller.

3 Design Details

We make the following assumptions when exploring opportunities for optimization:

- The link bandwidth is the bottleneck in the data center network. Many studies have shown this to be the case. For instance, Ballani et al. [6] have shown that 20% of network performance variability is caused by network bandwidth variance.
- The network has multiple shortest paths between pairs of servers.
- The network is heavily utilized at the core link layer due to network oversubscription. A study [8] by Microsoft and Berkeley stated that the core is heavily utilized in contrast to moderate utilization of the aggregation layer and top of rack layers.
- The network switches are programmable such that we can route packets using different paths between a pair of servers.

In the following sections, we describe the *least bottlenecked path* (LBP) algorithm for network layer path selection. We then present *network-aware caching* (NAC), our database layer optimization.

3.1 Least Bottlenecked Path

When there are multiple paths between pairs of servers, distributed systems typically use randomized algorithms, such as the *equal-cost multi-path* routing (ECMP), to perform load balancing. Proposals such as Hedera [3], mainly focus on load balancing for long-lived flows. Furthermore, these proposals typically require the controller to install dynamic network configurations in switches in response to every new flow. However, installing dynamic switch configurations is impossible for short-lived flows. By the time a new configuration is installed, the target flow may have already finished.

Least bottlenecked path (LBP) selects the least congested path among all the shortest paths between a pair of servers without installing new switch configurations. We consider only the shortest paths to avoid multiple traversals of core links since we assume that the core layer is oversubscribed. To determine the least congested path for a new flow, LBP first computes the maximum bandwidth each path can provide to the new flow. One reasonable approach, which provides fair sharing of the network resources between transactional and background flows, is to use global max-min fairness to compute flow bandwidth allocations. However, existing algorithms to achieve max-min fairness are computationally intensive and unsuitable for short-lived flows. Instead, we define the *link bandwidth factor* on a link as the value computed by dividing the capacity of the link by the expected total amount of data in transit on the link at a given moment. This link bandwidth factor is directional and can be used to approximate the available bandwidth from a source to a destination node on a particular link because the bandwidth available on is proportional to capacity and inversely proportional to the amount of data currently in transit.

As we have discussed in Section 2.2, the traffic on each link is captured and maintained by our controller. Furthermore, we identify that the bandwidth of a flow on a path is determined by the bandwidth of the bottleneck link on the path. Therefore, to determine the best path for a particular flow between multiple paths, we can compute the minimum link bandwidth factor on each path and pick the path with the largest link bandwidth factor. Algorithm 1 shows how we compute the minimum bandwidth link factor for a particular path. With link bandwidth factor information on all shortest paths, the controller can pick the path with the largest bandwidth by computing argmax_P ComputeBandwidthFactor(P) for each shortest path P between the source and destination.

Algorithm 2 We compute the link bandwidth factor for each link on the path. The minimum link bandwidth factor on a link models the maximum path bandwidth.

```
1:  procedure ComputerBandwidthFactor(path)
2:      MinBandwidthFactor = ∞
3:      for each link L in path do
4:          ▷ Expected bandwidth factor is the capacity of the link divided
            by the expected data in transit on a link.
5:          bandwidthFactor =
6:              L.BandwidthCapacity / (L.dataSize + 1)
7:          if bandwidthFactor < MinBandwidthFactor then
8:              MinBandwidthFactor = bandwidthFactor
9:          end if
10:     end for
11:     return MinBandwidthFactor
12: end procedure
```

Given the simple intuition behind our algorithms, the key is to implement LBP efficiently to handle short-lived flows. As mentioned in Section 2.2, NetStore configures the network switches to set up all paths when the system bootstraps. This avoids the overhead of installing new switch configurations at run-time. Since the controller uses DSCP bits in the TCP header to identify each unique path between every pair of servers, the controller piggybacks the DSCP values onto

the lock reply messages, which are sent back to the data servers. When a data server receives the lock reply messages, the data server will start performing every operation for which the lock is granted. For this reason, the controller can update flow count information as soon as it sends out the lock reply messages. Similarly, the controller updates flow count information when the lock release messages are received. The computation cost of Algorithm 1 is bounded by the maximum number of links between any pair of servers. For example, the maximum number of links in any path of three-tier multi-rooted tree is eight. Furthermore, the complexity of LBP is bounded by the number of unique paths between any pair of servers, which is likely to be small due to common data center network topologies.

To achieve horizontal scalability, LBP can easily work in a decentralized environment where each server collects and manages the global network state by querying OpenFlow [18] enabled switches in the datacenter network.

3.2 Network-Aware Caching

A previous study [7] has found that data center networks in shared environments are often heavily utilized at the core layer due to network oversubscription. Consequently, we can improve performance by reducing the load at the core layers. This is the intuition behind network-aware caching (NAC). To this end, we want to redistribute the core traffic to the aggregation or edge layers. Moreover, we identify that we can further reduce the network load by redistributing aggregation layer traffic to the edge. The simplest way to perform this redistribution is through standard caching mechanisms. In the case of our prototype, this means data servers can cache read operation results in local memory. In particular, many systems, such as MicroFuge [22], have shown that caching can improve system performance or help satisfy service level objectives. However, to provide consistency, cached copies must be either updated or invalidated when the corresponding data is overwritten. Data servers may need to invalidate multiple cache entries in the event of write operations. Communicating this evaluation on each write results in extra load on the network, hurting system performance. Instead, the controller can keep track of which data servers have temporary replicas of their read operation results and use this information to reduce network load.

To achieve this, we designed a two-layer caching scheme. At the center of the network, the controller has a local cache that keeps track of which replicas each data server has. At the edge, data servers maintain the replicated data. Moreover, we have augmented the controller to provide a version number for each replica. This version number increases each time we create a new cache replica. This versioning enables our system to provide an invalidation-free caching mechanism which avoids increasing the load on the network. Whenever the system performs a write operation on a key, the controller will invalidate this key's cache entry inside controller's metadata cache, which is described in Section 2.2. When this key is added back into controller's cache in the future, a new version number will be associated this cache entry. When a server performs a cache read at the edge of the network in data server, it will always check if the current version number provided by the controller matches the version number kept in the local cache of the data server. If the version number does not match, the data server will deem the cache copy outdated and fetch it directly from the server storing the original copy of the data. This enables consistency of the caching mechanism without introducing more load on the network.

Inside the controller, we have added one extra caching stage in the transaction lifecycle. We determine the server ID and the version number of the best replica, based on the network information, for all read operations in a transaction after the transaction has acquired all of its locks. This ensures consistency because any conflicting write operations cannot acquire a lock before read operations complete. If a cache entry exists for a particular read, the controller will determine the best replica and send this replica information along with the version number of the cache entry back to the data server. If the cache entry does not exist, the controller will insert this key into the cache with a new version number. This version is always larger than the previous version number because the server will automatically keep this item in its local cache after reading the data from the original server.

Recall that we want to minimize both the core link layer and the aggregation link layer traffic to redistribute the network load. Therefore, to determine the best replica, we have identified that only cross-pod operations will traverse the core link layer and only cross-rack operations will traverse the aggregation link layer. NAC is designed to reduce network load by reducing the number of cross-pod operations and number of cross-rack operations. For each cross-pod read operation `X`, when the controller is ready to send the lock reply message, the controller first looks into its local cache to see if a server in the same rack has a cached copy of the data. If the controller finds a cached copy in the same rack, the controller will ask the server to fetch the data from this replica. Otherwise, the controller will search for a cached copy in the same pod. If this fails as well, the controller will ask the server to fetch the data directly from the original server. We note that the controller does not need to suggest a local cache fetch because every server will always first look into its local cache to see if there is a cache hit. The load on the network will be reduced if the controller is able to turn cross-pod operations into same-rack or same-pod operations. The controller will also update its local cache to signal that this server has a cache replica for this key. A unique version number is generated only if this key is not previously stored in the controller's local cache.

On the server side, after a server receives the lock reply message, the server will first check to see if a cached copy is available with the matching version number for each read operation. If the version number does not match, the server will evict the item from the cache. To further reduce the network load, each server stores a list of keys whose data are currently being fetched. For each incoming cache request, where the server does not actually have a cached copy (i.e., it has been evicted), the server will first check if a duplicate read operation is already in flight. If so, it is beneficial to wait instead of sending an extra network message. After the fetch is completed, all waiting requests will be completed by the server with the latest data. If this fetch encounters outdated data from the cache server, the local server is now responsible to fetch the data from the origin server. This request will be filled after the second cache fetch is completed and the first cache fetch result will be sent back to the original requesting server. Both servers will keep a replicated copy of the data with the most recent version number.

Both the controller and data servers' cache sizes are system parameters which are provided when the system bootstraps. Those values can be tuned based on the physical resources available to the system. We implemented a clock-based eviction algorithm for the controller cache and we use random eviction for the data servers. Both eviction algorithms can easily be replaced based on the scenario. With this design, we can significantly reduce the amount of traffic generated by our transactional system while preserving consistency without implementing complex cache eviction schemes. To integrate NAC with LBP, the controller assumes that data servers fetch copies from the server suggested by the controller and the controller picks the best path using LBP for these network messages.

NAC can be made to scale horizontally with some distributed control. For instance, both decentralized locking and cache versioning can be achieved with a distributed consensus protocol. Furthermore, decentralized NAC nodes would not require communication for cache invalidation since our versioning scheme ensures only the current version of data is readable from the cache. We envision decentralized NAC implementation as a promising direction for future work.

4 Evaluation

In this section, we first describe our experimental setup. Then, we demonstrate the performance of NetStore by comparing least bottlenecked path and network-aware caching against *equal-cost multi-path* (ECMP) routing. In our comparisons, we will vary an array of different system parameters to demonstrate NetStore's performance.

4.1 Experiment Setup

To emulate a datacenter topology, we use Mininet [16] to build a distributed virtual network with a 1 Gbps capacity for each link. Rizvi, et al, use the same testbed to evaluate their distributed filesystem and a detailed description of this virtual network can be found in their paper [19]. In our testbed, we emulate a multi-rooted tree topology as discussed in Section 2 with 13 physical servers. Each physical server is a Supermicro SYS-6017R-TDF compute node consisting of 2 Intel E5-2620v2 CPUs, 64G RAM, 3 1TB SATA3 hard drives, 1 Intel S3700 200GB SATA-III SSD, 2 Intel i350 Gigabit Ethernet ports, and 1 Mellanox 10GbE SFP port.

In this evaluation, we use a modified version of ECMP as a baseline for comparison. To avoid TCP connection setup costs, our clients keep a persistent TCP connection with the local data server. For this reason, transactions do not have dynamic source and destination port numbers to perform traditional ECMP. To achieve ECMP-like behaviour, we use the same routing mechanism as NetStore, but randomly pick one of the unique shortest paths for each flow between a pair of servers.

In our emulation, there are 64 virtual machines. We run one transactional client, one background traffic client, one transactional server, and one background traffic server on each of the 64 virtual machines. Our background clients and servers are emulating network traffic in cloud environments. The volume of data center network traffic varies depending on many factors such as communication patterns of the deployed applications. This variation is captured in our experiments by controlling the max background data size as a tunable system parameter. We focus on cross-pod background traffic in our evaluation because the core layer links are often heavily utilized as discussed in Section 3.2. The background servers create multiple persistent TCP connections between every pair of data servers to emulate a realistic cloud environment.

By default, the transactional clients issue transactions at an interarrival rate that follows an exponential distribution with a mean value of 50 milliseconds. Each transaction either reads or writes 6,000 bytes of data to a randomly selected key. The key selection process follows a Zipfian distribution with a distribution constant of 0.99[3]. There are 1% write operations and 99% read operations in our workload. Each background client issues one background traffic request that follows an exponential interarrival rate with a mean value of one second. The background server transfers N bytes of data for each background traffic request, where N is uniformly selected from [1, Max background data size]. There are a total 2,000,000 unique key-value pairs stored in our 64 transactional data servers. The controller stores a maximum of 20,000 metadata cache entries and each data server may store up to 20,000 cache entries which accounts for 1% of all data entries in our data storage. Each experiment in our evaluation runs approximately 7 minutes. A summary of

[3]This is the same distribution constant used in YCSB [12].

default system parameters is listed in Table 4.1. These system parameters are empirically determined to demonstrate NetStore's performance in a partially congested datacenter network. We show the sensitivity of these parameters in the next section together with performance graphs where each data point on the graph is an average over 5 independent runs for every set of parameters.

NetStore parameter	default values
write transaction percentage	1%
foreground op data size	6KB
max background flow data size	8000KB
average fg interarrival time	50ms
average bg interarrival time	1000ms
# of key-value pairs in database	2,000,000
# of metadata cache entries in controller	20,000
# of cache entries in each data server	20,000

4.2 Performance of NetStore

The first goal of our experiments is to show that least bottlenecked path (LBP) routing can prevent transactional flows from colliding with background flows in the network, thereby improving system performance. Figure 4 compares the transaction completion times of ECMP with LBP. When the max background data size is at 8,000 bytes, LBP improves the average transaction completion time by about 47%. While at 12,000 bytes the network is heavily utilized, LBP can still achieve about 30% improvement for transaction completion times. This shows that LBP can effectively route transaction flows around background flows, reducing average transaction completion times when the network is the bottleneck. However, the performance of LBP is about the same as ECMP when the max background data size is 1,000 bytes because the network is not congested enough for LBP to take advantage of network information.

The second goal of our experiments is to demonstrate that network-aware caching (NAC) can reduce the load on the network, improving performance. In Figure 4, both ECMP and LBP are compared with the combination of LBP and NAC in terms of average transaction completion times. It shows that the combination of LBP and NAC can significantly improve the transaction completion times when the data size of the background flow is extended to 7,000 KB. In particular, NAC reduces average transaction completion time by 53% and 69% when the max background data size is at 11,000 bytes compared to LBP and ECMP respectively. The performance improvement increases with the maximum background data size because NAC effectively prevents performance deterioration caused by heavily congested network. This figure also shows that when the network is heavily congested, ECMP cannot load balance the system well and the transaction completion times increase significantly. In

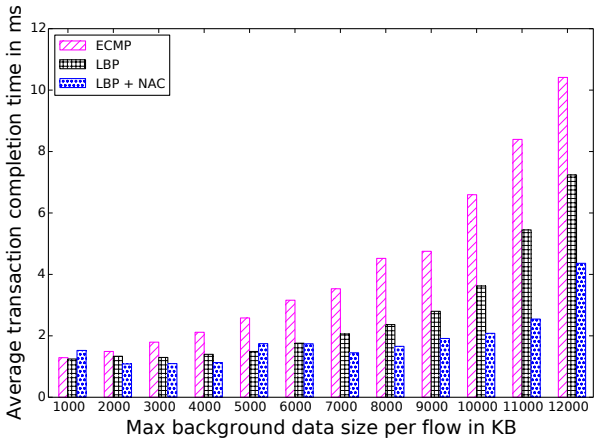

Figure 4. ECMP vs NetStore - varying the size of background flows.

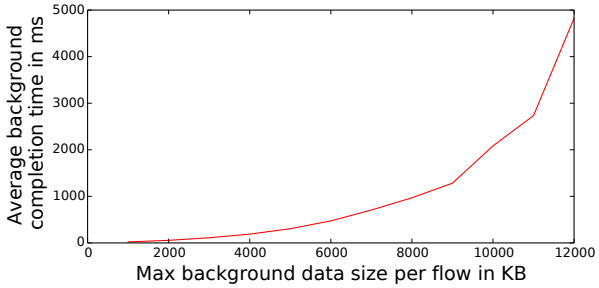

Figure 5. Average background flow completion time.

contrast, NetStore maintains a stable linear increase in the completion times with an increase in system load.

To understand the duration of each background flow, Figure 5 shows that when the max data size of a background flow is 1,000 KB, flows complete within 50 milliseconds. As we increase the maximum background flow size to 12,000 KB, flows will complete in 5 seconds on average. This shows that the range of the background flow data sizes covers the scenario from a network without congestion to a saturated network.

Next, we vary the number of operations in each foreground transaction to demonstrate NetStore's performance. Figure 6 shows that LBP and NAC outperforms by 48% and 84% when there are 5 operations because the increased number of operations improves both network-aware routing and caching opportunities. Note that the variance of the results has increased because a transaction completes after all of its operations have completed. With a larger number of operations, there is a higher probability that some operations within a transaction must traverse a congested part of the network which in turn increases the performance variance.

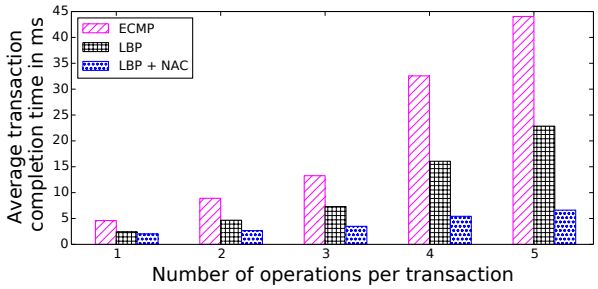

Figure 6. ECMP vs NetStore - Varying # of ops in transactions.

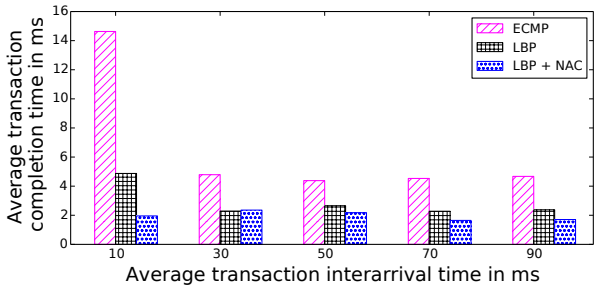

Figure 8. ECMP vs NetStore - varying average transaction interarrival time.

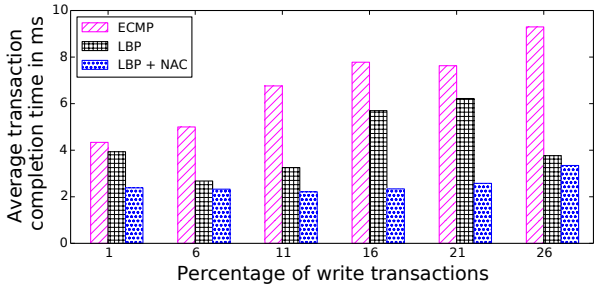

Figure 7. ECMP vs NetStore - varying write transaction percentage.

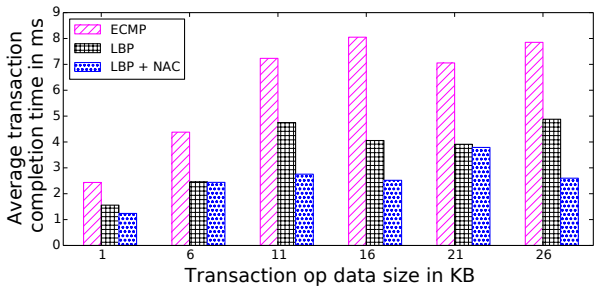

Figure 9. ECMP vs NetStore - varying op data size in transactions.

In Figure 7, we vary the percentage of write transactions in our microbenchmark. It shows that while transaction completion time increases as we increase the percentage of write transactions. The LBP and NAC performance improvement over ECMP is steady throughout the experiments. In the case of write-heavy workloads, we expect NAC to still provide a performance improvement for two reasons: (i) NAC introduces minimal overhead because the cache lookup operation inside the controller requires only a small number of hash table look-ups, and (ii) NAC implements a batch-processing mechanism whereby multiple consecutive reads of the same key from one rack will issue at most one cross-rack read request, reducing network load.

We vary the average transaction interarrival time in Figure 8, which also reveals that NetStore consistently performs better than ECMP. ECMP has a spike in response times when the average transaction interarrival time is at 10 ms because overwhelming foreground traffic may cause partial congestion in the network without network-aware routing algorithms.

Lastly, we vary the foreground transaction data sizes. Figure 9 shows that our algorithms have higher gains relative to ECMP when the foreground data sizes are large. However, as foreground data size gets smaller, the improvement comparing to ECMP diminishes because most operations complete in a short period of time and there is not much potential improvement.

The proceeding results demonstrate NetStore's superior performance compared to ECMP, showing that NetStore has promising potential to significantly reduce transaction completion times in a shared datacenter network.

5 Related Work

5.1 Network Systems

In the network research community, much effort has been invested in improving flow scheduling in datacenter networks. For example, DCTCP [5] has proposed a new transport layer protocol for datacenter networks. DCTCP uses Explicit Congestion Notification (ECN) combined with rate control to provide both high throughput for background traffic and low latency for short-lived flows, or "mice" flows. Alizadeh et al. [17] use priority scheduling in pFabric to prioritize mice flows to achieve lower average flow completion time. Alternatively, Wilson et al. [24] have proposed D^3, a deadline-aware network control protocol that targets mice flows to satisfy service level agreement (SLA) requirements. Unlike the aforementioned systems, we use a combination of load balancing and spatial locality to achieve better performance for short-lived transactions in the cloud.

Hedera [3] uses a centralized flow scheduler to increase total throughput. Unlike our work, Hedera targets bandwidth-intensive flows, or "elephant" flows, to reduce scheduling overhead. Moreover, Hedera needs to dynamically modify switch routing configuration at run-time, which is not suitable for mice flows in online transactional processing systems.

More recent work has identified that simple flow abstractions cannot capture all of the performance requirements of datacenter applications. Therefore, researchers have shifted their attention to solve the scheduling problem of coflows: tasks that consist of multiple flows. Dogar, et al., have proposed Baraat [14], a decentralized network scheduler. Similar to our work, Baraat does not require explicit switch coordination and it leverages limited-multiplexing to improve both the average completion time and tail latency. However, Baraat focuses on non-transactional tasks with multiple network flows such as web search. Our target is distributed transaction processing systems which provide strong consistency. Furthermore, Baraat models a simple network where there is only a single path between hosts.

Similar to Baraat, Chowdury et al. have proposed Varys [11] which uses the coflow abstraction to group multiple flows into a single entity for efficient scheduling. However, despite the performance improvement Varys can provide, the coflow abstraction cannot be easily used to implement database transactions.

Similar to NetStore, many of the aforementioned systems assume that flow size information is known a priori. Chowdury et al. recognize that most applications do not have such information. They have proposed Aalo [10], a centralized coflow scheduler that does not require flow size information a priori. We note that this is an interesting direction for our future work to estimate both the foreground and the background flow size information dynamically.

SquirrelJoin [20] focuses on network optimization for distributed join operations. Rupprecht, et al., identify that straggling workers resulting from transient network skew in shared clusters increases join completion time. They propose a lazy partitioning technique to mitigate this problem. In particular, SquirrelJoin delays a subset of partition assignment decisions to enable estimation and avoidance of network congestion dynamically. However, the distributed joins targeted by SquirrelJoin may take tens of seconds or more to complete, limiting the need for the rapid decision-making required in our low-latency environment.

5.2 Query Processing Systems

In the database community, the traditional approach to optimizing network usage in distributed query processing is to apply smart key placement strategies to minimize the number of distributed transactions, as well as the amount of data that must be transferred between nodes. For example, Schism [13] uses graph partitioning techniques to reduce the cost of distributed transactions by up to 30%. Vilaça et al. [23], target key-value stores and aim to improve locality in a multidimensional space of tags (such as foreign keys) applied to data items. We expect partitioning techniques such as these to be complementary to our work.

Xiong et al. [25] examine the usefulness of software-defined networking in supporting distributed analytical queries. Their workload consists of read-only SQL queries where query processing is bandwidth-intensive. They construct a global query optimizer which decides on join order and on how query results should be passed between sites. Similar to our work, their work targets distributed transaction processing. However, we focus on short-lived transactions instead of bandwidth-intensive transactions. This presents additional challenges since decisions must be made quickly. Moreover, we propose two novel optimizations in both the network layer and the database layer whereas their work focuses on integrating existing optimizations within the SQL query optimizer.

6 Conclusion

In this paper, we have made the case for network-level optimizations for cloud databases. These techniques leverage network-aware optimizations to support cloud applications without performance deterioration due to network saturation. In particular, we demonstrate the potential performance improvements by co-designing software systems with network optimizations. Our prototype system augments a distributed database system with a network manager which maintains network state information. Using this manager, we are able to apply least bottlenecked path (LBP), a network load balancing algorithm which intelligently routes requests to less congested parts of the network. Moreover, we introduce network-aware caching (NAC), a database layer caching technique, to further improve the system performance. Through experiments, we have shown that network-level optimizations can significantly reduce average transaction completion time.

Acknowledgments

This research is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Ontario Graduate Scholarship (OGS), the Queen Elizabeth II Graduate Scholarship in Science and Technology (QEII-GSST), and the David R. Cheriton Graduate Scholarship. The authors would also like to thank the anonymous reviewers for their insightful feedback.

References

[1] 2017. DSCP. https://en.wikipedia.org/wiki/Differentiated_services. (2017).

[2] 2017. Floodlight OpenFlow Controller. http://www.projectfloodlight.org/floodlight/. (2017).

[3] Mohammad Al-Fares et al. 2010. Hedera: dynamic flow scheduling for data center networks. In *NSDI'10*. Boston, USA.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1402958.1402967

[5] Mohammad Alizadeh et al. 2010. Data center TCP (DCTCP). In *SIGCOMM '10*. New Delhi, India.

[6] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *SIGCOMM '11*. Toronto, ON, Canada.

[7] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *IMC '10*. New York, NY, USA.

[8] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. 2012. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM '12*. Helsinki, Finland.

[9] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. 1991. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering* 17, 9 (Sep 1991), 954–960. https://doi.org/10.1109/32.92915

[10] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. In *SIGCOMM '15*. London, UK.

[11] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In *SIGCOMM '14*. Chicago, USA.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*. Indianapolis, Indiana, USA.

[13] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010).

[14] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. 2014. Decentralized task-aware scheduling for data center networks. In *SIGCOMM '14*. Chicago, USA.

[15] Namit Jain et al. 2010. Data warehousing and analytics infrastructure at Facebook in Proceedings. In *SIGMOD '10*. Indianapolis, IN, USA.

[16] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Hotnets-IX*. Monterey, USA.

[17] M. Alizadeh et al. 2013. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM '13*. Hong Kong, China.

[18] Nick McKeown et al. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR* 38 (2008).

[19] Sajjad Rizvi, Xi Li, Bernard Wong, Fiodar Kazhamiaka, , and Benjamin Cassell. 2016. Mayflower: Improving Distributed Filesystem Performance Through SDN/Filesystem Co-Design. In *ICDCS '16*. Nara, Japan.

[20] Lukas Rupprecht, William Culhane, and Peter R. Pietzuch. 2017. SquirrelJoin: Network-Aware Distributed Join Processing with Lazy Partitioning. *PVLDB* 10, 11 (2017), 1250–1261.

[21] Eric Schurmanand and Jake Brutlag. 2009. The user and business impact of server delays, additional bytes, and http chunking in web search. Velocity Conference Talk. (2009).

[22] Akshay K. Signh, Xu Cui, Benjamin Cassel, Bernard Wong, and Khuzaima Daudjee. 2014. MicroFuge: A Middleware Approach to Providing Performance Isolation in Cloud Storage Systems. In *ICDCS '14*. Madrid, Spain.

[23] Ricardo Vilaça, Rui Oliveira, and José Pereira. 2011. A Correlation-Aware Data Placement Strategy for Key-Value Stores. In *Distributed Applications and Interoperable Systems*, Pascal Felber and Romain Rouvoy (Eds.). Vol. 6723. Springer Berlin Heidelberg.

[24] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM '11*. Toronto, Canada.

[25] Pengcheng Xiong, Hakan Hacigumus, and Jeffrey F. Naughton. 2014. A software-defined networking based approach for performance management of analytical queries on distributed data stores. In *SIGMOD '14*. Snowbird, USA.