View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

Leveraging Software-Defined Networking to Improve Distributed Transaction Processing Performance

by

Xu Cui

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

© Xu Cui 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Recently, software-defined networking (SDN) has been transforming network technologies while NoSQL database systems are on the rise to become the de facto database systems for cloud technologies. Despite the promising progress in both worlds, there is little published work in bridging the two technologies. Moreover, even though numerous studies have reported that the network is often the performance bottleneck for cloud applications, network-aware database systems, which target the cloud environment, have not yet been explored.

In this thesis, we introduce NetStore, a new distributed transaction processing system that bridges the gap between network research and distributed database research to avoid transaction performance deterioration due to network saturation. NetStore leverages the SDN technology with both network layer and database layer optimizations to support transaction processing with network-awareness. In particular, with the help of the SDN controller, NetStore is able to apply a novel load balancing algorithm at the network layer. Moreover, NetStore introduces a database layer optimization to redistribute network load. In addition, a transaction scheduler, that relies on the performance model of the underlying system, is also introduced to further improve the system performance. Our experiments have shown that NetStore can reduce the average transaction completion time by as much as 78% while doubling the system throughput.

Acknowledgements

I would like to thank my supervisors, Professor Bernard Wong and Professor Khuzaima Daudjee, for their helpful advice and patient guidance. I would also like to thank Professor Paul Ward and Professor Ken Salem for being my thesis readers and for their insightful feedback. I would also like to thank Michael Mior for his contribution to this project. I would also like to thank Sajjad Rizvi for helping me with the network emulator setup. I would also like to thank Nicole Keshav for helping me correct grammar errors in this thesis.

My parents and my girlfriend have always supported me in every aspect of my life. I would not be able to finish this thesis without them.

Dedication

This is dedicated to my parents Jingnian Cui and Dongmei Liu.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the increasing proliferation of big data, companies need to process terabytes of new data on a daily basis [23]. To manage data at this scale, non-relational distributed databases, such as MongoDB [22], Redis [3] and Cassandra [24], have emerged recently. Non-relational databases typically expose simple 'get' and 'put' operations and delegate query optimization to the application. Query optimization is critical to reduce application response time. Numerous studies have shown that delays in milliseconds can have significant impact on business revenue [26] [32] [16]. However, some query optimizations, such as network layer optimizations, are difficult to perform for distributed databases hosted within datacenters. Due to the shared nature of datacenters, applications neither have a global view nor control of resources. This query optimization problem is especially difficult when the datacenter network is the bottleneck because applications usually do not have any control over the datacenter network. To this end, there is a rapidly growing technology called *Software-Defined Networking* (SDN) [27]. SDN enables software programs to perform network management and provisioning using a logically centralized controller. In this thesis, we propose a new transactional database called NetStore which combines the high performance of non-relational databases with the network control capability of SDN.

NetStore is designed to leverage network-awareness to provide high performance transaction processing capabilities in the event of high network utilization. In its design, SDN provides a logical centralized controller. This enables us to monitor and manage network traffic, which is crucial for datacenter applications. The NetStore system consists of an SDN controller and a distributed non-relational database. The centralized SDN controller serves as a centralized lock server that is required to provide consistency and isolation guarantees. The SDN controller also collects network states and manages network flows with its transaction scheduler. The transaction scheduler sits on top of the SDN controller and schedules queries based on current network conditions. Through the integrated system architecture, NetStore provides performance optimizations from network layer to application layer.

1.1 Problem Overview

Key-value storage systems often shard data into multiple partitions that can be stored on different servers. Application programmers can perform application level optimization based on their data access to achieve the best performance. However, one of the main problems facing cloud storage systems is that the network is increasingly the performance bottleneck for many cloud applications. The cloud providers may overcome this bottleneck by over-provisioning the network bandwidth in the system. However, this comes at a cost of wasted resources. This waste is especially problematic for cloud environments because cloud networks exhibit On/Off behaviours [10] [9]. That is, the network traffic spikes from time to time, which may cause network links to be saturated. Another option is to dynamically turn resources on or off. However, this kind of service is likely not available to datacenter tenants; and even if it were, it would be very difficult for cloud consumers to determine the exact time to turn on or off a particular server. Furthermore, transaction processing systems, which are used by many applications, are one of the most response time sensitive components deployed in datacenters. For this reason, we believe that re-architecture of database systems is required to better use and manage the network resources. To this end, we have designed NetStore which uses SDN to greatly reduce the system response time even when the network is saturated. To understand the problem in depth, we explain the details of datacenter network in the following section.

1.2 Datacenter Network Architecture

Datacenter networks come in many forms. We model the datacenter network using one of the most common datacenter network topologies, the multi-rooted tree.

Figure 1.1 shows a typical multi-rooted tree topology with two core switches. A pod is a discrete unit of datacenter components, which consists of a number of interconnected servers and switches. In Figure 1.1, there are two pods each consisting of two aggregation switches, four top-of-the-rack (ToR) switches, or edge switches, and eight servers. Each ToR switch is connected to each aggregation within its pod and different pods are connected using links between core switches and aggregation switches. Typical datacenter network

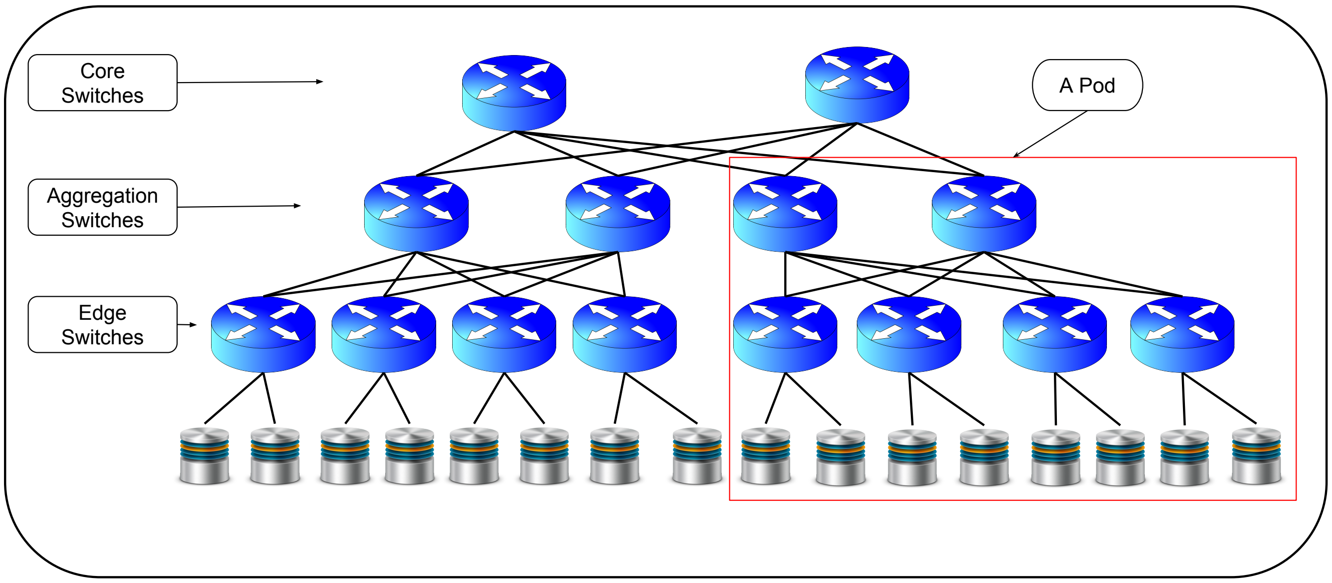

Figure 1.1: A Multi-rooted Tree Topology

architectures will have oversubscription[1] at each layer of the network. For example, in our experiments, we set up topology such that the Core to Aggregation links have an oversubscription factor of 2, i.e., the link bandwidth ratio is 2:1 for core links versus aggregation links. The aggregation to edge links have an oversubscription factor 5. This gives the network a total subscription of 10:1 from servers to core switches.

There are eight different paths between two servers if they are from different pods. Even within the same pod, there are four paths in total, between any two servers that do not reside within the same rack. Traditional systems use *Equal-Cost Multi-Path* (ECMP) to select a path between two hosts. The ECMP algorithm uses the hash values of a five-tuple, consisting of source IP, destination IP, source port, destination port, and protocol type to select one of the shortest[2] paths between a pair of servers. However, this selection is oblivious to network conditions such as available bandwidth.

[1]An oversubscription is a situation in which the output link bandwidth is lower than the input link bandwidth in a network switch or a network layer.

[2]Shortest means least number of hops.

1.3 NetStore

NetStore is a network-aware transactional key-value storage system. It has three major components: The storage backend is a distributed transaction storage system which is built on top of LevelDB [21]. The core of NetStore, the network controller, is built upon extensions to the Floodlight [2] software-defined networking controller. The controller is responsible for setting up network routes among sources and destinations, as well as monitoring the network conditions to see how many flows are currently using a particular route. On top of the network controller, we have a scheduler which is responsible for scheduling each coordinator's request based on the current system states. The controller keeps track of the number of flows on each link within the system to make informed decisions. A client will simply communicate with one of the NetStore dataservers. The contacted dataserver will act as a coordinator for a given transaction and perform the actions and inform the client of the results or the outcome. NetStore provides simplified transaction APIs to the client, but these APIs are sufficient to implement many transactional applications such as RUBiS [12].

1.4 Contributions

The main contributions of this thesis are as follows:

- *Least Bottlenecked Bandwidth* (LBP), a novel load balancing algorithm that selects the least congested path among all of the shortest paths between a pair of servers for each network flow
- *Opportunistic Load Redistribution* (OLR), a network-aware application layer technique that uses temporary replicas to reduce the load on saturated network links
- *Earliest Expected Job First* (EEJF), an intelligent scheduler that uses a network-aware performance model of the underlying system to improve the performance of OLR by delaying network flows that are likely to traverse congested links

1.5 Organization

The rest of this thesis is organized as follows: Chapter 2 surveys related work. Chapter 3 presents an overview of the system architecture and Chapter 4 discusses the design details of NetStore. Chapter 5 presents experimental setup and results. Chapter 6 concludes the thesis.

Chapter 2

Related Work

As NetStore combines software-defined networking with cloud storage systems, this chapter will survey the related work from these two areas. Following an overview of SDN in Section 2.1, Section 2.2 surveys existing work on query optimization techniques for cloud storage systems and Section 2.3 surveys prior work on network flow scheduling.

2.1 Software-Defined Networking

As an overview of SDN technologies, traditional networks evolved around the principle to keep network logic simple. Switches cannot be easily programmed by application programmers. Therefore traditional applications treat networks as a black box. We have been witnessing advancement in Internet technology over the past two decades. However, network technology has not advanced much because of the ossification of the Internet [36]. For instance, people cannot easily replace the Internet protocol to prototype new techniques because too much business logic and profit rely on the Internet to function. SDN has been introduced to solve this ossification problem. SDN separates network control from data transportation to provide application programmers a high level interface to modify the network dynamically to meet their specific needs. This enables system architecture to easily prototype applications with network-aware properties and make new design decisions based on network information. SDN allows network architectures to program only part of the network while keeping old functionalities working. This is especially useful for network researchers who want to experiment without breaking existing infrastructure. SDN views the network as two separate planes: the control plane and the data plane. The sole purpose of the data plane is to ship data from its source to destination based on control plane decisions. The controller acts like the brain of the network, it can decide which path a flow must take from point A to point B. The controller may ask the switch to drop certain packets based on certain conditions.

2.2 Query Optimization

The majority of work on query processing in key-value storage systems is on executing queries on data stored in distributed hash tables (DHTs) in peer-to-peer (P2P) networks. For example, Harren, et al. [19], propose a basic query model for executing complex queries in this context. Their approach aims to minimize communication cost, but it is designed for P2P systems where the network is not under control of the application and is therefore unable to make use of network-aware techniques. PIER [20] is a similar query processing engine over DHTs which is also not network-aware.

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 [15] uses graph partitioning techniques to reduce the cost of distributed transactions by up to 30%. Vilaça, et al. [37], specifically 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.

Similar to our work, CloudTPS [39] provides a query processing layer on top of NoSQL stores which enforces ACID semantics for transactions. However, instead of focusing on performance improvements, CloudTPS focuses on providing ACID guarantees in the event of network partition or server failures. Rödiger, et al. [31], examine the effect of data locality with respect to distributed query processing. Their Neo-Join algorithm is designed for efficient network usage by repartitioning data and scheduling network transfers to avoid cross-traffic. These techniques do not utilize control over the network, which may provide further opportunities for optimization.

For example, Xiong et al. [41], 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 NetStore, this work targets distributed transaction processing. However, NetStore focuses on short-lived transactions instead of bandwidth-intensive transactions. Moreover, NetStore proposes novel optimizations in both the network layer and the database layer whereas this work only focuses on enhancing existing optimizations within the SQL query optimizer.

There are other advanced query operations which may benefit from software-defined networking. Ntarmos, et al. [29], uses novel indexing techniques to improve the performance of top-k joins for NoSQL databases, which, in turn, reduces both network latency and traffic. This work did not explicitly use software-defined networking. However, the amount of network traffic is a key metric in their evaluation, hinting at the possibility to successfully apply SDN techniques.

2.3 Network Flow Scheduling

Much effort has been invested in improving flow scheduling in datacenter networks. For example, DCTCP [6] 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. [7], use priority scheduling in pFabric to prioritize mice flows to achieve lower average flow completion time. Alternatively, Wilson, et al. [40], 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, NetStore uses a combination of load balancing and spatial locality to achieve better performance for short-lived transactions.

Hedera [5] uses a centralized flow scheduler to increase the throughput of the network links. Similar to NetStore, Hedera targets multi-rooted tree topologies where there may be multiple equal cost paths between a pair of servers. However, unlike NetStore, 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 abstraction cannot capture all of the performance requirements of datacenter applications. Therefore, researchers have shifted their attention to solve the scheduling problem of coflows or tasks that consist of multiple flows. To this end, Sparrow [30] is a decentralized scheduler that targets sub-second jobs. To achieve decentralization without compromising performance, Sparrow uses a randomized load balancing technique. It combines the power-of-two-choices technique [28] with batch processing and late binding to perform accurate scheduling with limited decentralized information. By contrast, NetStore takes a step further to improve the performance of sub-second transactional systems. NetStore maintains a global view of the network by augmenting the centralized lock server to the SDN controller.

More recently, Dogar, et al., have proposed Baraat [17], 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-transaction tasks with multiple network flows such as web search. NetStore is a distributed transactional database system that provides ACID properties. Furthermore, Baraat models a simple network where there is only a single path between any hosts.

Similar to Baraat, Chowdury, et al., have proposed Varys [14] which uses coflow abstraction to group multiple flows into a single entity. Varys models datacenter networks as a single switch. Varys has proposed two heuristics, Smallest-Effective-Bottleneck-First (SEBF) and Minimum-Allocation-for-Desired-Duration(MADD), to schedule coflows efficiently. However, despite the performance improvement Varys can provide, the coflow abstraction cannot be easily translated into database transactions. In particular, Varys' techniques will require major revisions to provide ACID semantics to coflows. Unlike NetStore, which is built to solve this problem.

Many aforementioned systems require flow size information a priori. Chowdury, et al., recognize that most applications do not have such information. They have proposed Aalo [13], a centralized coflow scheduler that does not require flow size information a priori. Aalo applies Least-Attained-Service [8] to schedule coflows with a discretized scheme to improve average completion time. Aalo is most similar to our work, while NetStore takes the solution one step further to target the flow scheduling problem in a transactional setting, without requiring detailed knowledge of each transaction.

To conclude, Table 2.1 summarizes the characteristics of the flow scheduling techniques we have discussed in the chapter.

System Name	Decentralized Design	Dynamic Switch Configuration	ACID Semantics	Elephant Flow	Sub-Second Jobs	Multiple Flow Support	Flow Size Knowledge	Multiple Paths between Hosts
pFabric	✓				✓		✓	
D^3	✓				✓		✓	
Hedera		✓		✓			✓	✓
Sparrow	✓				✓	✓		
Baraat	✓			✓	✓	✓	✓	
Varys				✓		✓	✓	
Aalo				✓		✓		
NetStore			✓	✓	✓	✓		✓

Table 2.1: A Summary of Flow Scheduling Studies.

Chapter 3

System Architecture

NetStore is a key-value storage system that processes transactions issued by clients. Our transaction model provides a slightly expanded interface over the common "get" and "put" operations used in similar systems [18] [3]. These additional operations allow developers to implement non-trivial transactional applications. In this chapter, we first describe our transaction model and system APIs in Section 3.1. The architecture and components of NetStore are introduced in Section 3.2. We describe the details of our software-defined networking (SDN) controller in Section 3.3. Finally, we present our distributed data storage server in Section 3.4.

3.1 Transaction Model

NetStore's transaction model provides atomicity, isolation, consistency, and durability. A modified version of the two phase commit protocol (2PC) is used to provide atomicity. The modified 2PC allows participants to send the PREPARED message to the coordinator with the READ/WRITE result in a single message. This technique saves one round-trip-time (RTT) for the 2PC protocol, as proposed by Aguilera, et al. [4]. To provide isolation, NetStore relies on a centralized lock server. Each operation within a transaction is required to acquire a READ or WRITE lock before execution. After a transaction is completed, it will release all of the locks it holds. This strict two-phase-locking (2PL) combined with 2PC provides strict serializability to NetStore. The NetStore dataservers use the "get" and "put" operations provided by LevelDB [21] to interact with the persistent storage of the underlying system, which in turn provides durability.

NetStore transactions consist of a number of operations in which each operation works with a single key-value pair. The NetStore operations within a transaction are fully independent of each other, which gives us the freedom to reorder operations within a transaction and allows us to increase the level of concurrent during transaction executions. However, by removing these dependencies, it becomes hard to implement certain useful applications. Therefore, along with basic READ and WRITE operations, NetStore also provides INCREMENT/DECREMENT and APPEND operations to data, which eliminates some common sources of data dependencies within a transaction. Also provided are operations which will conditionally abort a transaction by checking if a given key exists or by comparing the value of a key with a predefined value. Transactions with read-modify-write semantics can use this mechanism and multiple transactions to implement the same logic via compare-and-set with retries, as necessary. Furthermore, transactions are considered completed when all of the operations within a transaction are completed and the results of each operation within the transaction are sent back to the client.

Table 3.1 shows operations that are supported by NetStore. A single transaction consists of one or more operations. A transaction is considered to be successful if and only if all of its operations are completed successfully. If one of the operations is aborted, the entire transaction is also aborted. Each operation consists of a tuple of type, key, and value.

Operation Name	description
READ	read data
WRITE	overwrite data
APPEND	append to data
COMPARE_EQ	integer comparison(equality), abort if not true
COMPARE_GREATER	integer comparison(greater), abort if not true
COMPARE_GREATER_EQ	integer comparison(greater or equal), abort if not true
COMPARE_LESS	integer comparison(less), abort if not true
COMPARE_LESS_EQ	integer comparison(less or equal), abort if not true
INCREMENT	integer increment
DECREMENT	integer decrement
KEY_EXISTS	check if a key exists in db, abort if not true
KEY_NOT_EXISTS	check if a key does not exist in the db, abort if not true

Table 3.1: NetStore Operation Types

3.2 System Architecture Overview

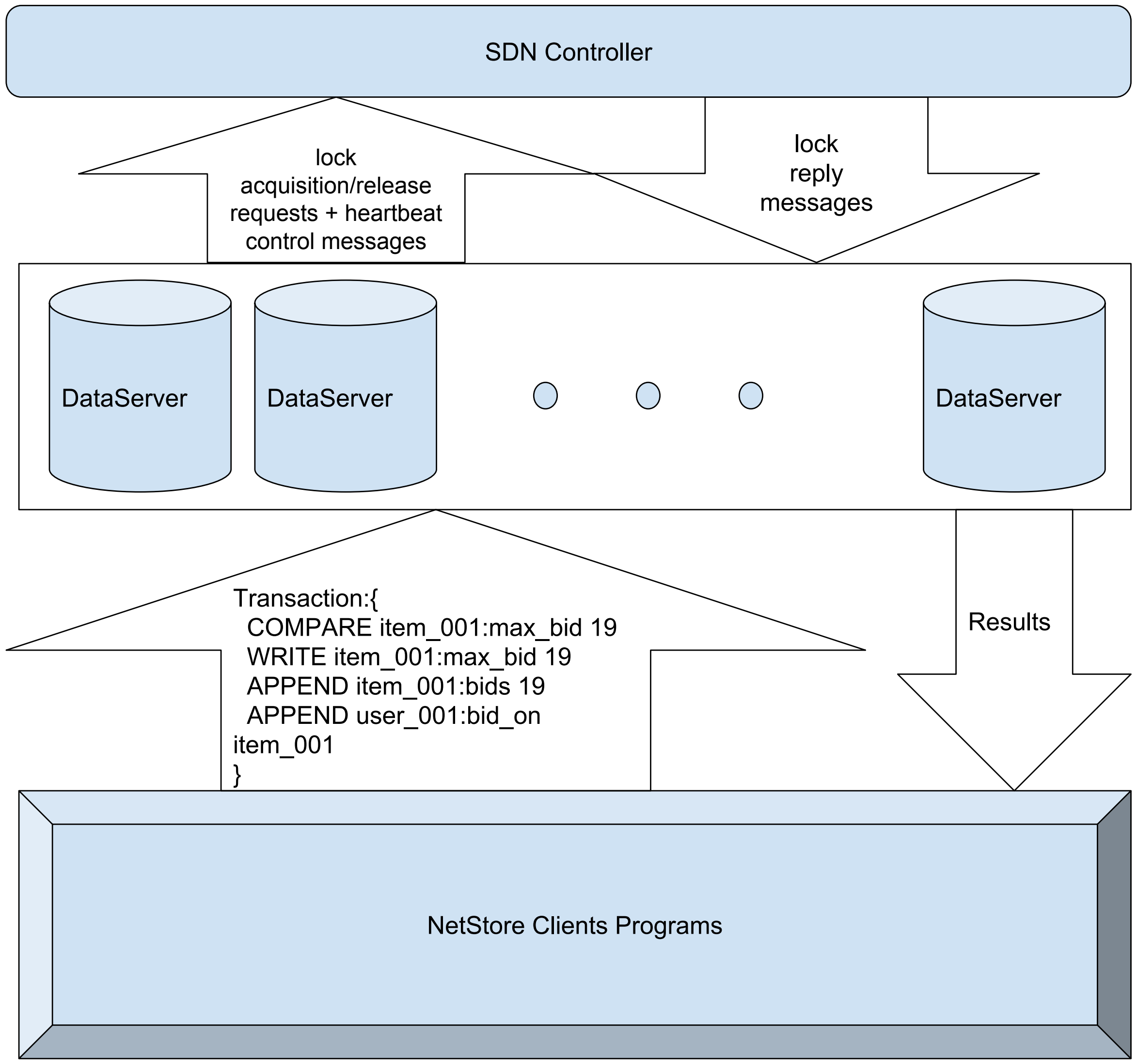
SDN Controller
lock acquisition/release requests + heartbeat control messages
lock reply messages
DataServer
DataServer
DataServer
Transaction:{
COMPARE item_001:max_bid 19
WRITE item_001:max_bid 19
APPEND item_001:bids 19
APPEND user_001:bid_on item_001
}
Results
NetStore Clients Programs

Figure 3.1: NetStore Architecture

NetStore consists of a centralized lock server, one or more distributed database servers and a client library for users to develop applications. To leverage software-defined networking (SDN) technology, we augmented the SDN controller to handle lock requests. To that end, this allows our system to piggyback many control messages onto the lock related messages.

As shown in Figure 3.1, the client programs can interact with NetStore using NetStore's transactional APIs. Clients can aggregate multiple operations in Table 3.1 and issue all of the operations as a single transaction. A client can send its transactions to only the nearest dataserver in the network topology. This design allows NetStore to have maximum control over the internal network of the system. The dataserver receiving the query will act as the coordinator for this particular transaction. The coordinator will contact the controller to acquire the necessary locks, perform each operation, and send the final result back to the client. The controller will grant a READ/WRITE lock to each operation within a transaction, while maintaining the lock states. Since the dataserver can only perform an operation after a successful acquisition of a lock, the controller can effectively act as a scheduler by controlling when to grant the lock to the transaction coordinator. In addition to lock control and scheduling, the controller also determines which network path each operation should take. The path information is piggybacked onto the controller's messages to coordinators. Upon receiving the lock acquisition message from the controller, the coordinator will in turn perform the operations by interacting with its local storage or with a remote dataserver.

The core of NetStore is the controller which is built as an extension to the Floodlight SDN controller. This centralized controller not only acts as a centralized lock server, but it also monitors the network state of the system. Based on the global system state, the controller can make intelligent scheduling decisions.

3.3 Controller

As the core of NetStore, the controller performs four crucial tasks. Figure 3.2 shows that the controller not only acts as a centralized lock server, but it also acts as the brain of the entire system.

First, the controller implements a standard READ/WRITE lock server where a READ lock is a shared lock and a WRITE lock is an exclusive lock. All of the transaction operations can be classified as READ or WRITE operations, e.g. a key existence check is essentially a read operation and an increment is equivalent to a write operation. The controller keeps updated information about lock acquisitions and releases. To avoid deadlocks,

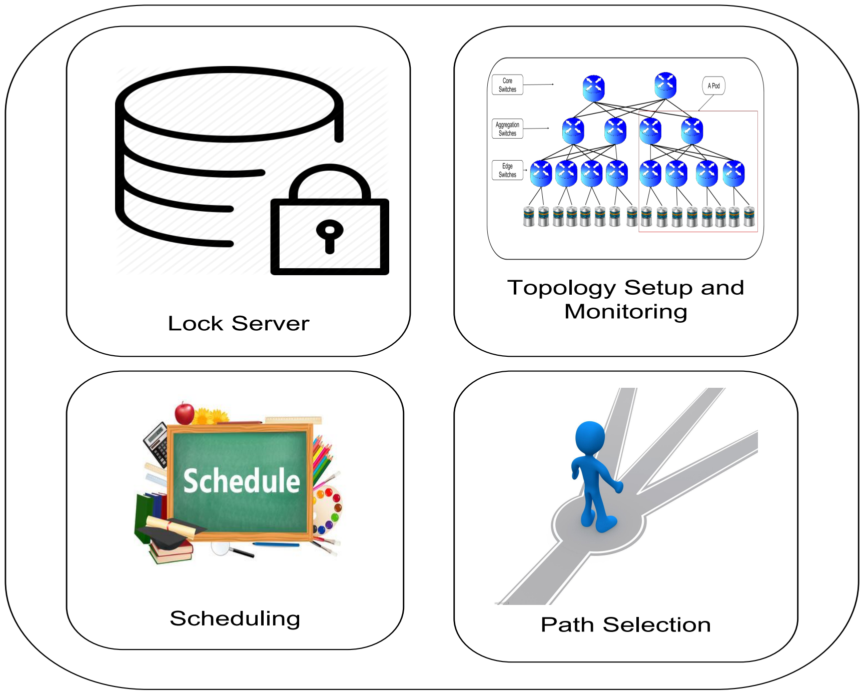

Figure 3.2: NetStore's Controller Functions

the controller implements a first-in-first-out (FIFO) lock queue to grant lock permissions to individual transactions. This means that, if there are two transactions with conflicting lock requests, the request which arrives earlier will always acquire the lock first. With the strict-two-phase-locking (S2PL) implementation at the dataserver, this guarantees that no cycles can exist in the wait graph. Therefore, deadlock is avoided in NetStore.

Second, the controller also serves as a network topology manager. When the controller bootstraps, the controller will read topology information such as link bandwidth from configuration files. Moreover, the controller will discover all of the switches and links in the topology through the link-discovery logic provided by the Floodlight SDN controller. The controller aggregates all of this information to build a virtual topology, which the controller can use to set up path configurations between each pair of servers. In particular, if there are multiple paths between a pair of servers, the switch configuration is required to distinguish each unique path. To that end, the controller will configure the switches such that, in addition to the source and destination pair, the DSCP [1] bits in the TCP headers will also be used to identify a unique path between a pair of servers. Furthermore, the controller will update metadata of the virtual topology to track the number of flows on each link at any given time. The overhead for these updates is minimal because they are performed when the controller grants or revokes locks in response to lock acquisition or release requests.

With the virtual topology, path information and monitoring ability, the controller is able to make intelligent path selections and scheduling decisions to greatly improve the performance. The details are discussed in Chapters 4 and 5.

3.4 Dataserver

The distributed dataserver is based on LevelDB [21]. LevelDB provides simple 'get' and 'put' operations to interact with persistent storage. To service transactional queries, the NetStore dataserver expands this simple abstraction to provide enhanced APIs, as listed in Table 3.1. Furthermore, each dataserver is also responsible for distributed communication with other dataservers as well as the controller. For instance, when NetStore bootstraps, each dataserver will establish one or more TCP connections with every other dataserver. The number of TCP connections depends on the number of unique shortest paths between a pair of servers. To execute a transaction, the assigned dataserver (coordinator) must perform all of the operations in the transaction. Furthermore, for each remote operation, while the dataserver is required to use the path selected by the controller, the dataserver is oblivious to the network topology, because it only needs to modify the DSCP value in the TCP header based on the information sent by the controller. This design significantly reduces the complexity of the dataserver and leaves all of the network control logic at the controller. Another crucial responsibility of the dataserver is to provide ACID semantics to transactions as discussed in Section 3.1.

Figure 3.3 depicts the lifecycle of a transaction. So far, we have discussed every stage in the transaction lifecycle except the two red boxes. In the next chapter, we will examine how NetStore achieves better performance when presenting the design detail for our SDN controller.

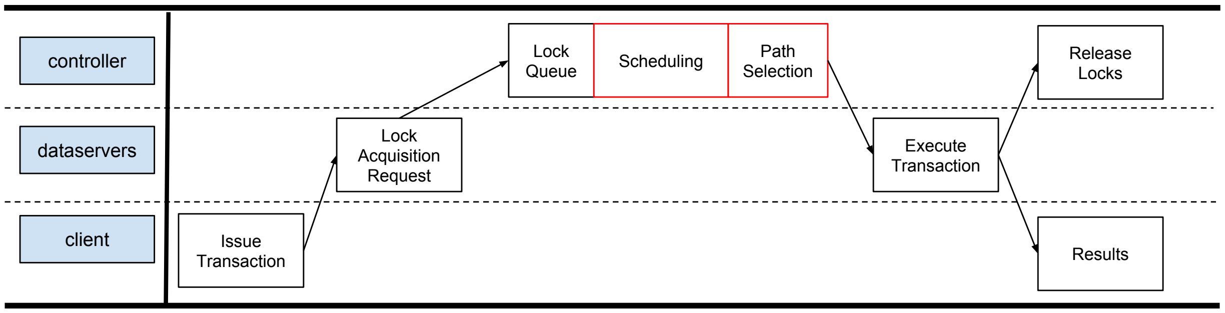

Figure 3.3: Transaction Lifecycle

Chapter 4

Design Details

In this chapter, we describe our design assumptions followed by the design goal. We then explain how our SDN controller is designed to significantly improve the performance of transaction execution.

We make the following assumptions in designing our system:

- The datacenter network is the bottleneck for executing transactional queries.
- The datacenter network has multiple shortest paths between pairs of servers.
- The datacenter network is heavily utilized at the core link layer.
- The network switches are programmable such that we can route packets using different paths between a pair of servers.

Our design goal is to reduce average completion times of transactions and increase system throughput. To achieve these goals, we apply optimizations to different system layers of NetStore. In particular, we combine network layer optimization with database layer optimization to improve the overall system performance.

In the following sections, we first describe the motivation and the details of the *Least Bottlenecked Path* (LBP), our network layer path selection algorithm. We then present the *Opportunistic Load Redistribution* (OLR), our database layer optimization. Last, we explain the details of the *Earliest Expected Job First* (EEJF) algorithm, which is designed to further improve the performance of OLR.

4.1 Least Bottlenecked Path

In this section, we first describe the motivation for why LBP is necessary to improve transaction processing performance. Then, we describe the design details of LBP followed by a description of the implementation details.

4.1.1 Motivation

When there are multiple paths between pairs of servers, traditional systems usually use randomized algorithms, such as *Equal-Cost Multi-Path* routing (ECMP), to perform load balancing. Recent proposals, such as Hedera [5], mainly focus on load balancing for long-lived flows. Furthermore, these proposals usually require the controller to install dynamic network configurations in switches in response to every new long-lived flow. However, the dynamic switch configurations are impossible for short-lived flows. For instance, by the time a new configuration is installed, the target short-lived flow may have already finished. We believe that flow schedulers need an efficient solution for the load balancing of short-lived flows. In this thesis, we propose LBP to solve this problem.

4.1.2 Design Details

LBP selects the least congested path among all of the shortest paths between a pair of servers. We consider only the shortest paths to avoid multiple traversal over the core links because we assume that the core link 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 popular approach is use the global max-min fairness algorithm to compute flow bandwidth allocations. However, this algorithm is computationally intensive for short-lived flows. To simplify the complexity of online computations, we assume that all of the flows on a link share the bandwidth equally. For instance, if there are two flows on a link with 2 Mbps of bandwidth, we assume that each flow gets 1 Mbps bandwidth.[1] Since the bandwidth of a flow in a path is determined by the bottleneck link bandwidth of the path, we can compute the path bandwidth available to a new flow using the number of existing flows on each link. As we have discussed in Section 3.3, this information is captured and maintained by our controller when the locks are acquired or released by the transactions. Therefore the controller can use Algorithm 1 to predict the

[1]We want to note that this is a reasonable assumption for short-lived flows, which are likely the dominant flows in transaction executions, but this not true for long-lived flows.

bandwidth of a new flow on each possible path . With the bandwidth information, the controller can pick the path with the largest bandwidth using Algorithm 2. Given the simple intuition behind our algorithms, the key is to implement LBP efficiently to handle short-lived flows. Implementation details are described in the following subsection.

Algorithm 1 ComputeBandwidth

```
1: procedure ComputeBandwidth($path$)
2:     $MinBandwidth = Double.MAX$, $rtn$
3:     for each link $L$ in $path$ do
4:         $bandwidth = L.BandwidthCapacity$ / $(L.CurrentNumberOfFlows + 1)$
5:         if $bandwidth < MinBandwidth$ then
6:             $MinBandwidth = bandwidth$
7:             $rtn = bandwidth$
8:         end if
9:     end for
10:    return $rtn$
11: end procedure
```

Algorithm 2 LBP algorithm

```
1: procedure LBP($src$, $dst$)
2:     $MaxBandwidth = Double.MIN$, $rtn$
3:     for each shortest path $P$ between $src$ and $dst$ do
4:         $bandwidth$ = ComputeBandwidth($P$)
5:         if $bandwidth > MaxBandwidth$ then
6:             $MaxBandwidth = bandwidth$
7:             $rtn = P$
8:         end if
9:     end for
10:    return $rtn$
11: end procedure
```

4.1.3 Implementation Details

The design goal of LBP is to perform load balancing of flows efficiently and accurately. As mentioned in Section 3.3, NetStore configures all of its paths when the system bootstraps.

This avoids the overhead for installing new switch configurations at run-time. Since the controller uses DSCP bits in the TCP headers 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 dataservers. When a dataserver receives the lock reply messages, the dataserver 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. However, extra work is required to accurately update the flow count information when a flow finishes. Recall that NetStore uses *strict-two-phase-locking* (S2PL); this implies that the lock release messages cannot accurately reflect when a particular flow is finished. Therefore, the NetStore dataserver needs to send one extra heart-beat message, when a remote operation is completed, to inform the controller to update the flow count information. Last, the computation cost of Algorithm 1 is bounded by the number of links between any pair of servers. For instance, the max number of links, in any path, is eight for a three-tier multi-rooted-tree topology. Furthermore, the complexity of LBP is bounded by the number of unique paths, which is likely to be small, between any pair of servers.

4.2 Opportunistic Load Redistribution

In this section, we will first describe the motivation behind *Opportunistic Load Redistribution* (OLR). We will then discuss the design and implementation of OLR.

4.2.1 Motivation

In our design assumptions, we assume that the network is heavily utilized at the core link layer. Consequently, we can improve the performance of NetStore by reducing the network load at the core link layer. This is the basic intuition behind OLR. To this end, we want to redistribute the core link layer traffic to the aggregation layer or to the edge layer. The simplest way to perform this load redistribution is through standard caching mechanisms. For instance, the NetStore dataserver can cache READ operation results in its local memory. In particular, many systems, such as MicroFuge [34], have shown that caching can help to improve system performance or help to satisfy Service Level Objectives. However, to provide ACID semantics, much coordination is required for distributed dataservers. For instance, dataservers may need to invalidate multiple cache entries in the event of WRITE operations. This coordination may result in extra load to the core link layer, which may hurt the system performance. For this reason, NetStore takes a new approach. We identify that the controller can determine which dataservers should keep temporary replicas of their READ operation results to reduce the core link layer traffic because the controller acts as both the lock server and the network manager. Moreover, we have designed the NetStore controller such that it is able to determine how many times the temporary replica of a READ result will be read by subsequent operations. This enables NetStore to perform load redistribution without sacrificing the ACID semantics or without introducing more load to the core link layer. This design detail is covered in the next subsection.

4.2.2 Design of OLR

Our goal is to allow the controller to determine if the temporary replica of a READ operation will be read by future operations. To this end, we need the controller to be able to inspect future operations. The simplest approach is through limited multiplexing of transactions. Therefore, we have augmented the NetStore controller with a pending queue. The controller will allow only a limited number of concurrent operations in the system by artificially delaying lock reply messages. This limit is a system parameter which can be set when the system bootstraps. If the controller detects that the number of concurrent operations has reached this limit, the NetStore controller will put future transactions, which have acquired all their operation locks, into the pending queue. Therefore, we have added one extra stage in the transaction lifecycle. A transaction may enter the pending queue after it has acquired all of its operation locks. This design has three advantages. First, limited-multiplexing allows NetStore to withhold transactions when the network is congested. Second, the controller has full flexibility in reordering transactions for performance reasons because no two conflicting transactions can enter the pending queue at the same time. Last, all of the READ operations, for the same key, in the pending queue will always read the same data because the WRITE operation for the same key is in conflict with these READ operations. Therefore, our controller can perform opportunistic look-ahead and determine if it is beneficial to temporarily store a READ result in dataserver memory.

Equally important, we have identified that only cross-pod operations will traverse the core link layer. Therefore, the opportunistic look-ahead aims to reduce the number of cross-pod operations. To do that, for each cross-pod READ operation X, when the controller is ready to send the lock reply message, the controller looks into the pending queue to determine if there are any other cross-pod READ operations for the same key. Furthermore, the controller will only look for the cross-pod READ operations whose source dataservers are within the same pod as the source dataserver of operation X. The load on the core link layer will be reduced if these qualified operations can read data from the source dataserver of operation X. Therefore, if there are one or more qualified operations, the source dataserver of operation X will keep a temporary replica of the read result. Moreover, the controller can also determine exactly how many times this temporary replica will be read. The NetStore dataserver will maintain a counter for each temporary data replica. This counter will be decremented upon each read. When the counter reaches zero, the dataserver will automatically delete the item from its memory.

With this design, we know that a temporary data replica will only exist while there are one or more transactions holding the READ lock of this data item. No write operation can be performed on the same key while the READ lock is being held. For this reason, NetStore preserves ACID semantics without implementing complex cache eviction schemes.

4.2.3 Implementation Details

To implement OLR, the controller is augmented with a pending queue for transactions. After a transaction has acquired all of its locks, the transaction will enter the pending queue. The controller will try to execute as many transactions from the pending queue as possible with respect to the number of concurrent operations allowed in the system. As mentioned in the preceding subsection, before the controller sends out the lock reply message to a dataserver in response to a READ operation, the controller will scan every operation in the pending queue to determine if it is beneficial to keep the read data result in memory. On the receiving side, the dataserver will perform the remote operation and store the retrieved data item in a hashtable in memory. For each item in the hashtable, the dataserver also keeps a counter for that item. When a counter reaches zero, the associated item will be deleted from the hashtable. The controller is also responsible to inform those selected operations that they must read from a different destination dataserver. To this end, the controller will also piggyback the new destination information onto the relevant lock reply messages.

In summary, our OLR implementation does not generate any new flows in the network because all of the new information is piggybacked onto lock reply messages.

4.3 Earliest Expected Job First

In this section, we will discuss the motivation, design and implementation of the *Earliest Expected Job First* algorithm (EEJF).

4.3.1 Motivation

OLR can help to reduce the load of the core link layer, which in turn improves the system performance. In particular, we find that the temporary replication of larger data items, on a congested link, has bigger impact on the system performance than the temporary replication of smaller data items.[2] For this reason, we designed EEJF to help OLR further reduce the number of larger flows that traverse congested core layer links.

4.3.2 Design Details

To accurately identify large flows that traverse congested core layer links, we have built a performance model of the underlying system in the NetStore controller. Intuitively, if a transaction takes a long time to complete, it suggests that the transaction contains large flows that traverse congested core links. Consequently, our performance model uses historical data to predict the completion times of transactions. The historical data is a window of past operation completion times maintained by the controller. To translate the raw historical completion times into a usable performance model, we note four key observations. First, a transaction's completion time is likely to be determined by the slowest operation in the transaction. Second, the operations can be classified by the types of data items that each operation reads or writes. For example, some data types may suggest that the data items are likely to be larger than others. Third, the operations with the same data type can be further classified by the location of their source and destination dataserver pairs. For instance, cross-pod operations are likely to take longer than those operations which can be performed within a rack. Finally, even though there are multiple paths between pairs of servers, LBP has already ensured that the unique paths between a pair of servers are evenly loaded. Therefore, we do not further classify operations in terms of unique paths.

For these reasons, the NetStore controller uses a hashtable to store historical data in the form of completion times. The key of the hashtable consists of the data type and the source-destination pair of an operation. Since NetStore focuses on scheduling both long-lived and short-lived flows, we believe that part of the historical information is also short-lived. Thus, instead of keeping a large window of past history, for each operation class in the hashtable, the controller keeps only the last completion time for each operation class.

[2]The details are presented in Section 5.4 in Chapter 5

Algorithm 3 PredictTransactionCompletionTime

```
procedure PredictTransactionCompletionTime($t$, $PerformanceModel$)
    $rtn = 0$
    for each operation $op$ in $t$ do
        $src = op.src$
        $dst = op.dst$
        $type = op.type$
        $opCompletionTime = PerformanceModel$.get($src$, $dst$, $type$)
        if $rtn < opCompletionTime$ then
            $rtn = opCompletionTime$
        end if
    end for
    return $rtn$
end procedure
```

With the help of the performance model, the controller is able to predict transaction completion times using Algorithm 3. This algorithm enables the NetStore controller to improve performance by reordering transactions in the pending queue. The simplest approach is to use *Shortest Job First* (SJF). By delaying the transactions with longer predicted completion times, OLR is more likely to serve the operations, which contain large flows and traverse core layer links, from replicated data storage. However, SJF may lead to starvation which is not desired for *Online Transaction Processing* workloads (OLTP). Therefore, we propose EEJF, a new scheduling algorithm that avoids starvation. Like SJF, EEJF gives each transaction a priority and inserts the transaction into the pending queue based on this priority value. However, instead of using absolute priority values, EEJF sets the priority of each transaction as the expected transaction completion time. To this end, the controller prioritizes shorter flows while the priority of the longer flows will increase as time passes. This avoids both starvation and complex computations.

4.4 Implementation Details

To implement EEJF, we need to change OLR's *first-in-first-out* (FIFO) queue to a priority queue. Consequently, when EEJF is turned on, the controller will insert transactions into the priority queue based on the priority value of each transaction. Extra effort is required to build the performance model of the underlying system. However, except for the extra time required to set up the hashtable of the performance model when the system bootstraps, the cost of updates is very low. For instance, the NetStore dataserver piggybacks the operation data type information onto the lock request messages. Furthermore, the controller is aware of the operation start times because an operation can only start after receiving the controller's lock reply message. Moreover, the controller is also aware of the operation end times because the dataservers send heart-beat messages to update the controller's flow count information when an operation completes. Therefore, the update of historical completion times adds only a little overhead to the system.

This concludes the design details of NetStore. We will present the evaluation of NetStore in the next chapter.

Chapter 5

Evaluation

5.1 Experimental Setup

In this section, we first describe the workload that is used to evaluate NetStore. We then describe the details of our emulated datacenter topology, which is used in our experiments.

5.1.1 Workload

Our workload is based on the RUBiS [12] OLTP benchmark. RUBiS simulates an online auction website where users view, comment, and bid on a list of items. RUBiS was originally implemented as a Java web application backed by a MySQL database. To adapt RUBiS to our purposes, we have implemented simple clients that execute the necessary transactions against NetStore to simulate the same workload. Our clients issue transactions in a closed loop with probabilities matching the distribution from the original RUBiS implementation. In RUBiS, a single request is in the form of an interaction. Each interaction consists of one or more transactions. Furthermore, each interaction retrieves all of the data required to serve a single web page in the original RUBiS benchmark. For this reason, our measurements focus on the throughput, the completion time, and the tail latency of the interactions. Table 5.1 shows the different interactions and their corresponding probabilities. Approximately 7% of the transactions issued by this workload involve writes.

Interaction	Write?	% of total
Register User	✓	1.01
Browse Categories		6.89
Search Category		15.23
Browse Regions		2.05
Search Region		5.93
View Item		14.18
View User Info		3.17
View Bid History		1.85
Buy Now		1.39
Store Buy Now	✓	1.36
Put Bid		5.86
Store Bid	✓	4.31
Put Comment		0.53
Store Comment	✓	0.52
Register Item	✓	0.53
About Me		2.66

Table 5.1: RUBiS Interaction Types

5.1.2 Network Topology

To emulate a datacenter topology, we use Mininet [25]. Mininet allows us to emulate a multi-rooted tree topology with multiple virtual servers in a single host server. The host server, in our testbed, is a Supermicro SSG-6047R-E1R26L Large compute node consisting of 2 Intel E5-2630v2 CPUs, 256G RAM, 14 2TB 7200RPM SAS2 Hard drives, 1 Intel S3700 400GB SATA3 SSD, and 1 Intel P3700 400GB PCIe NVMe solid-state storage device. As shown in Figure 5.1, our topology consists of eight virtual servers. Each virtual server runs one NetStore dataserver to handle client requests. We shard the entire RUBiS data into eight equal segments and store each segment in one of the dataservers. We run multiple copies of RUBiS clients on each virtual server. We varied the number of clients in our experiments. In particular, each virtual server may have 20 to 100 clients running concurrently. This amounts to 160 to 800 clients in the entire system.

A summary of the experimental setup is shown in Table 5.2. In particular, the link bandwidth between the core switches and the aggregation switches is 30Mbps; the link bandwidth between the aggregation switches and the edge switches is 150Mbps, and the link bandwidth between the edge switches and the virtual servers is 300Mbps. Note that the link bandwidths are limited because Mininet can run only within a single host server.

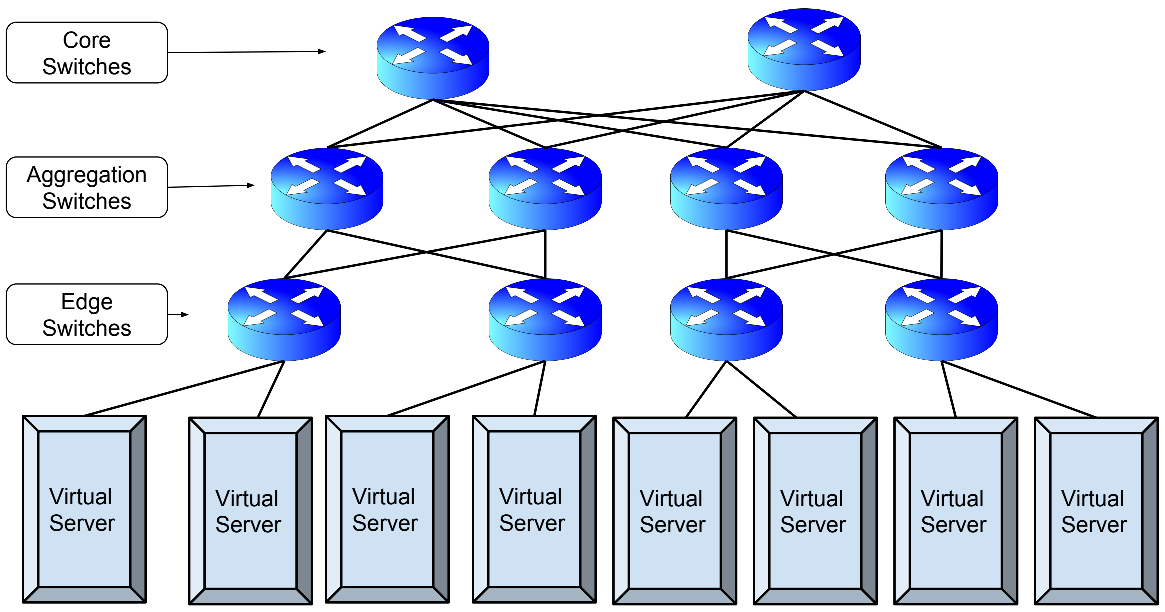

Figure 5.1: Testbed Network Topology Setup

Mininet uses CPUs to switch virtual network packets. For this reason, the link bandwidths in the testbed are chosen to cope with the CPU limitation of a single host server. Moreover, in a typical multi-rooted tree topology, the aggregation layer switches and the aggregation layer links usually have higher bandwidth than the edge layer switches and the edge layer links. However, there are usually hundreds or thousands of edge layer switches, connected to racks of servers, in a typical datacenter. To reduce the switch and link costs at the aggregation layer, the aggregated input bandwidth of the aggregation layer links is usually lower than the aggregated input bandwidth of the edge layer links. Similarly, to reduce the switch and link costs at the core layer, the aggregated input bandwidth at the core layer is usually lower than the aggregated input bandwidth at the aggregation layer. For this reason, our emulated testbed has an oversubscription ratio of 10:5:1 from the edge layer to the core layer.

In this evaluation, we use a modified version of *Equal-Cost-Multi-Path* routing (ECMP) as a baseline for comparison. The traditional ECMP algorithm uses the hash values of a five-tuple, consisting of source IP, destination IP, source port, destination port, and protocol type to select paths for each flow. However, our clients issue transactions in a

Environment Parameter	Value
Benchmark	RUBiS
Number of Key Value Pairs (Data Records)	32,211,000
Edge Link Bandwidth	300 Mbps
Aggregation Link Bandwidth	150 Mbps
Core Link Bandwidth	30 Mbps
Average Think Time	500 ms
Number of Servers	8
Number of Clients	160 - 800
Number of Runs	5
Ramp Up Time	90 seconds
Measure Time	540 seconds
Ramp Down Time	90 seconds

Table 5.2: Experimental Setup Configuration

closed loop. This means that, a client will continuously issue transactions, one at a time, for the entire duration of an experiment. To avoid TCP connection setup costs, our clients keep a persistent TCP connection with the local dataserver. For this reason, transactions do not have dynamic source and destination port numbers to perform traditional ECMP. To achieve ECMP-like behaviour, we use *Round-Robin* (RR) to route flows when there are two or more unique paths between a pair of servers.

In the following sections, we present the experimental results of NetStore. For each experiment, we have performed five independent runs. The error bars represent the 95% confidence interval over the 5 runs. Furthermore, we aggregate all of data in 5 runs for each experiment to determine the 95th and the 99th percentiles. For the purpose of brevity, we mostly focus on describing the results for 480 clients, which is the median of our client range.

5.2 Performance of Least Bottlenecked Path

The first goal of our experiments is to show that the *Least Bottlenecked Path* (LBP) can provide better load balancing in the network, which in turn improves system performance.

Figure 5.2 compares the interaction completion times of ECMP with LBP. At 480 clients, LBP improves the interaction average completion time by 25%, While at 800 clients

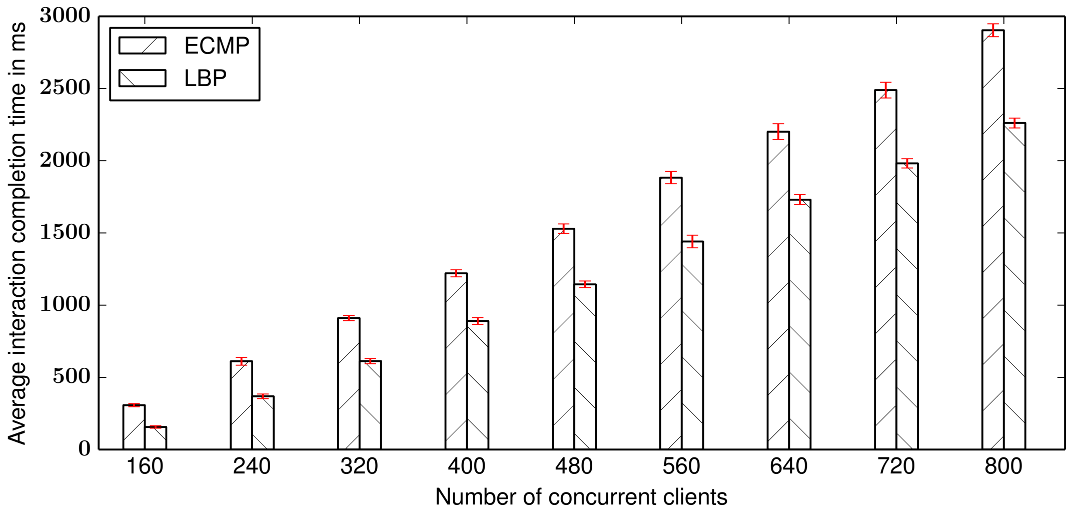

Figure 5.2: ECMP vs LBP - Average interaction completion time.

the network is heavily utilized, LBP can still achieve about 22% improvement for completion times. This shows that LBP can achieve better load balancing of the network with the help of accurate flow count information on every link in the topology, which in turn reduces average interaction completion times when the network is the bottleneck.

Figure 5.3 compares the interaction throughput of ECMP with LBP. LBP consistently outperforms ECMP by more than 20% when there are 240 or more clients in the system. In particular, LBP improves the system throughput by 21% at 480 clients. This confirms that, in a closed system, better average completion times lead to better throughput.

Next, we examine the results of the 95th and 99th percentile completion times. In particular, we focus our analysis on three interactions in the RUBiS benchmark. The *ViewUserInfo* interactions consist of read operations for data items with a size of 12.5 bytes; the *StoreBid* interactions consist of mixed read and write operations; and the *BrowseCategories* interactions consist of read operations for data items with size of 125 kilobytes. We have chosen these three interactions because they represent three interaction categories in the RUBiS benchmark. In particular, these categories are read-only interactions, read-write interaction, and relatively data-intensive interactions.

The graphs in Figure 5.4 and Figure 5.5 show that LBP outperforms ECMP by as

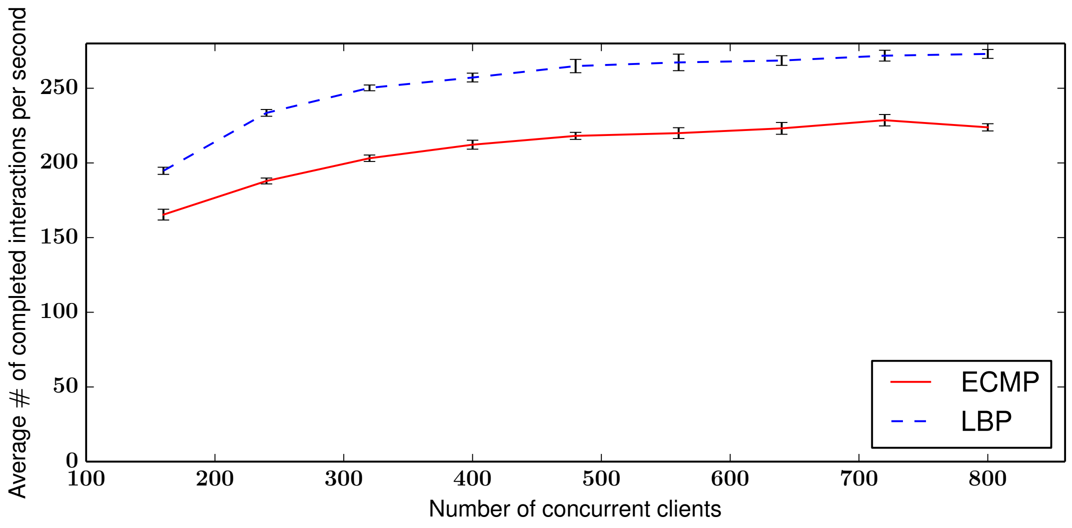

Figure 5.3: ECMP vs LBP - Throughput

much as 46%, for the percentiles of ViewUserInfo interactions, when the system is not heavily loaded. However, as the number of clients increases, this improvement eventually disappears. This is because the benefit of load balancing diminishes as the network links become saturated.

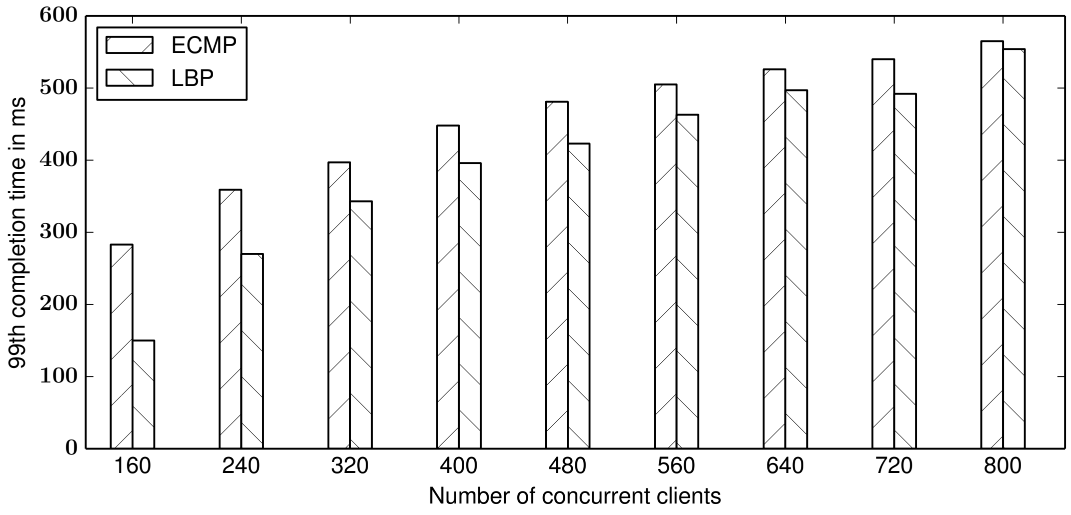

Figure 5.4: ECMP vs LBP - 95th percentile interaction completion time for ViewUserInfo

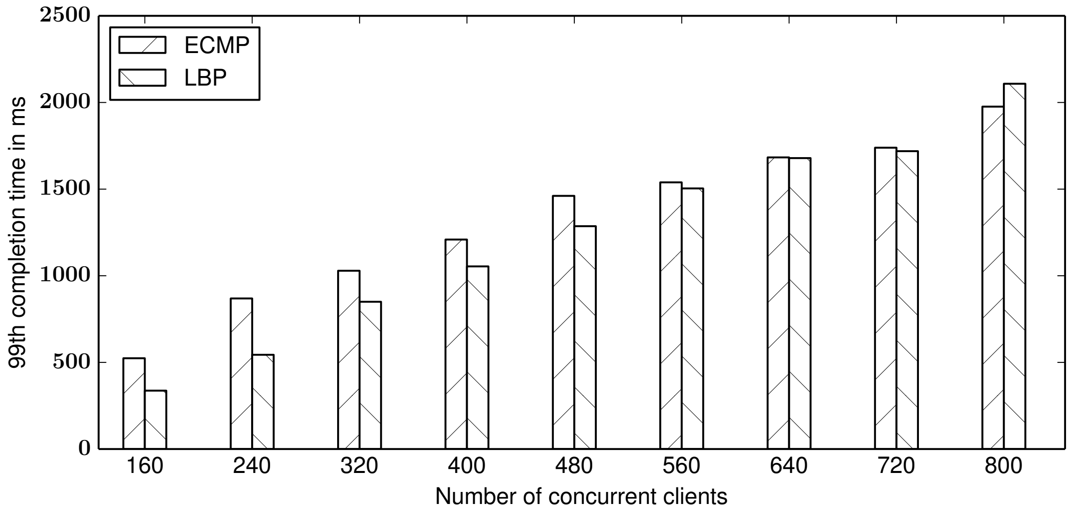

Figure 5.5: ECMP vs LBP - 99th percentile interaction completion time for ViewUserInfo

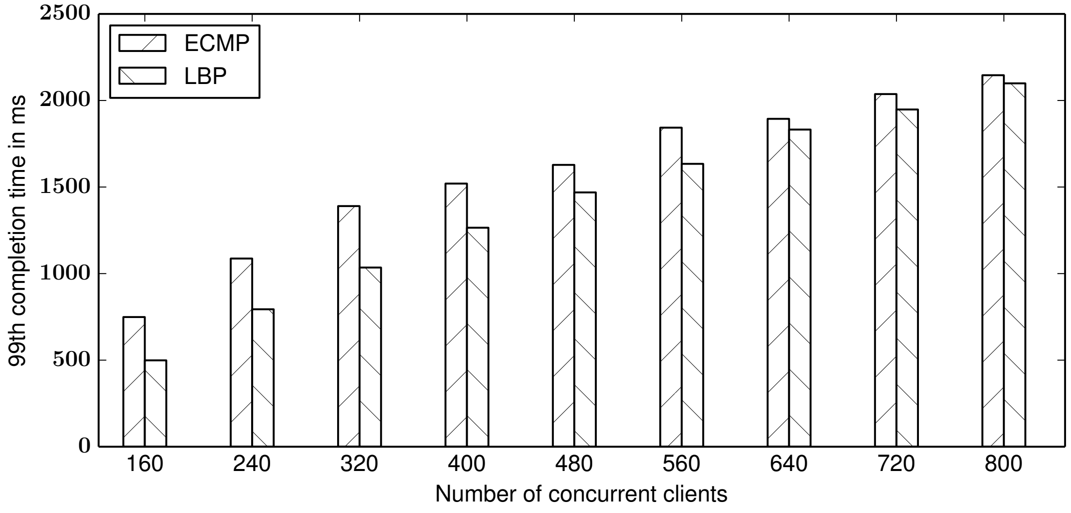

Figure 5.6: ECMP vs LBP - 95th percentile interaction completion time for StoreBid.

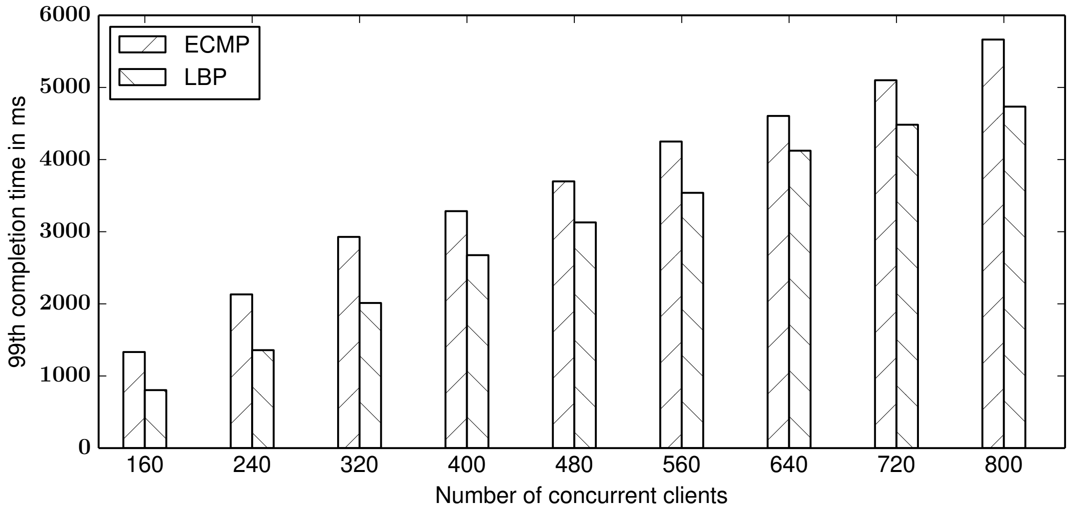

Figure 5.7: ECMP vs LBP - 99th percentile interaction completion time for StoreBid.

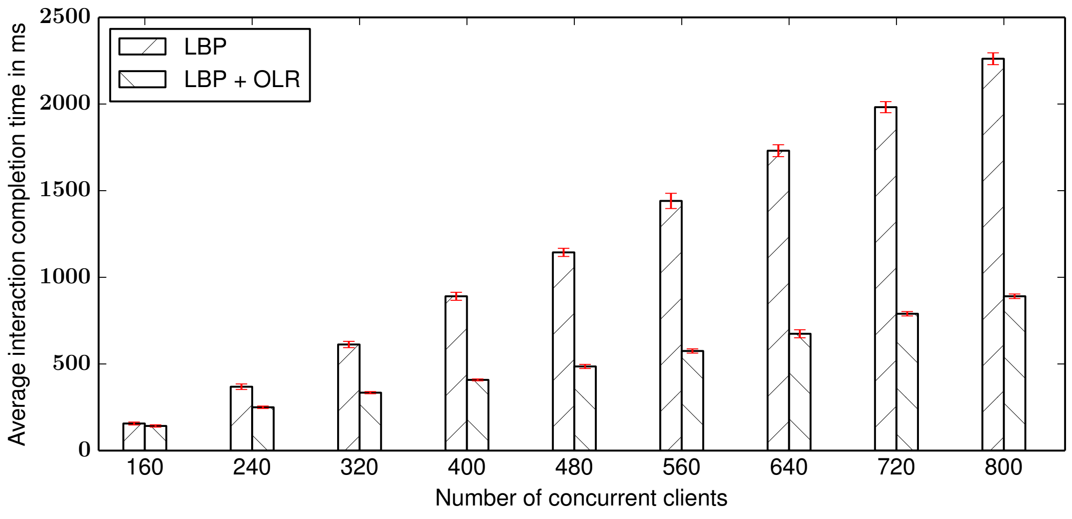

Figure 5.8: LBP vs (LBP + OLR) - Average interaction completion time.

The 95th and the 99th percentile results for StoreBid interactions are shown in Figure 5.6 and Figure 5.7. We see that LBP consistently outperforms ECMP for the 99th percentile results. This is because the extra complexity of StoreBid interactions, compared to the ViewUserInfo interactions, leaves more space for improvement in terms of load balancing even if the system is overloaded. Similarly, consistent improvement in tail lantencies is also observed for flows in the BrowseCategories interactions. Thus, these graphs are omitted for brevity.

5.3 Performance of Opportunistic Load Redistribution

The second goal of our experiments is to demonstrate that the *Opportunistic Load Redistribution* (OLR) can reduce the load on core link layer. In this section, LBP is compared with a combination of OLR and LBP in term of the average interaction completion times, the throughput, and the percentile results.

Figure 5.8 shows that OLR can significantly improve the average interaction completion times. In particular, OLR has reduced the average interaction completion time by 57% for 480 clients. Furthermore, the performance improvement has reached about 60% when there are 800 clients. This is because as the number of clients increases, the length of the pending queue in the controller will also increase. For this reason, OLR has more opportunities to perform load redistribution, which in turn provides better performance. However, the performance of OLR is almost same as LBP when there are only 160 clients because there are not enough transactions in the pending queue to benefit from load redistribution.

Similarly, as shown in Figure 5.9, OLR improves the throughput of the system by 57% for 480 clients. Moreover, OLR provides almost twice of the throughput that LBP can achieve when there are 800 clients. This result shows that OLR can effectively reduce the load on the core link layer in a datacenter network, which in turn increases the system throughput. We want to note that the extra memory required to cache the temporary replicas is bounded by the size of the pending queue in the controller. Moreover, through measurements in all of our experiments, we find that the total extra memory required, counting all of the dataservers, is less than 1 megabyte in size, which is less than 0.5% of the total data size in the RUBiS benchmark.

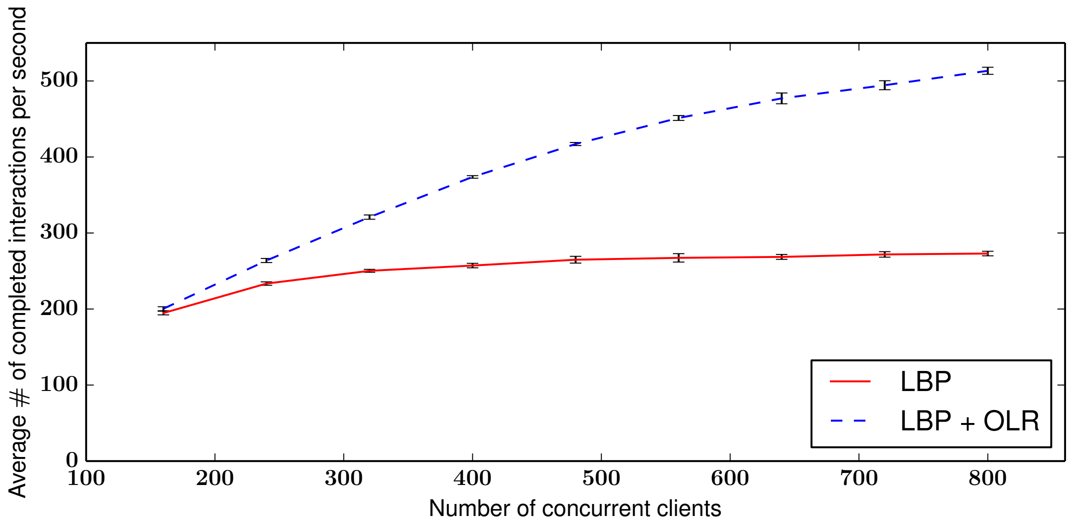

Figure 5.9: LBP vs (LBP + OLR) - Throughput.

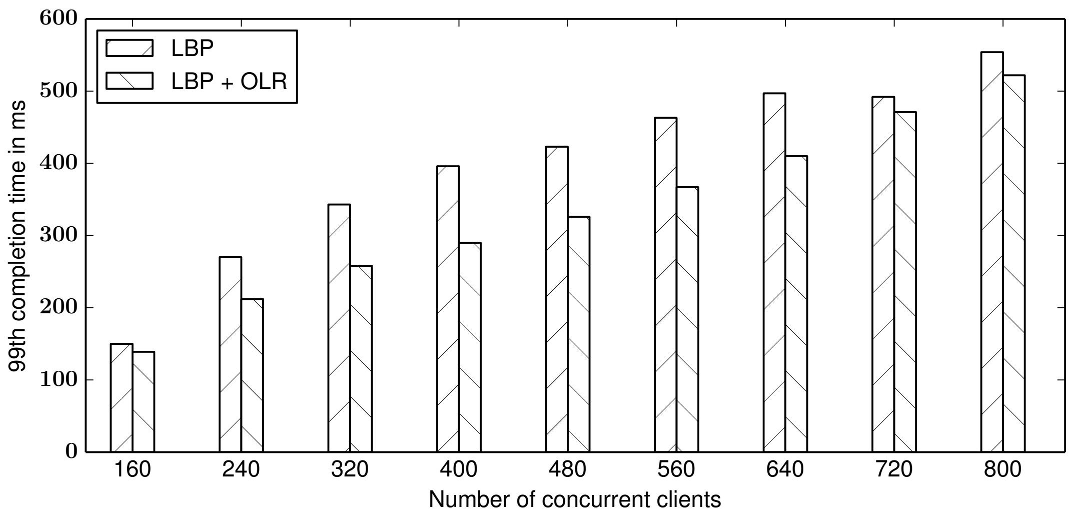

Figure 5.10: LBP vs (LBP + OLR) - 95th percentile interaction completion time for ViewUserInfo.

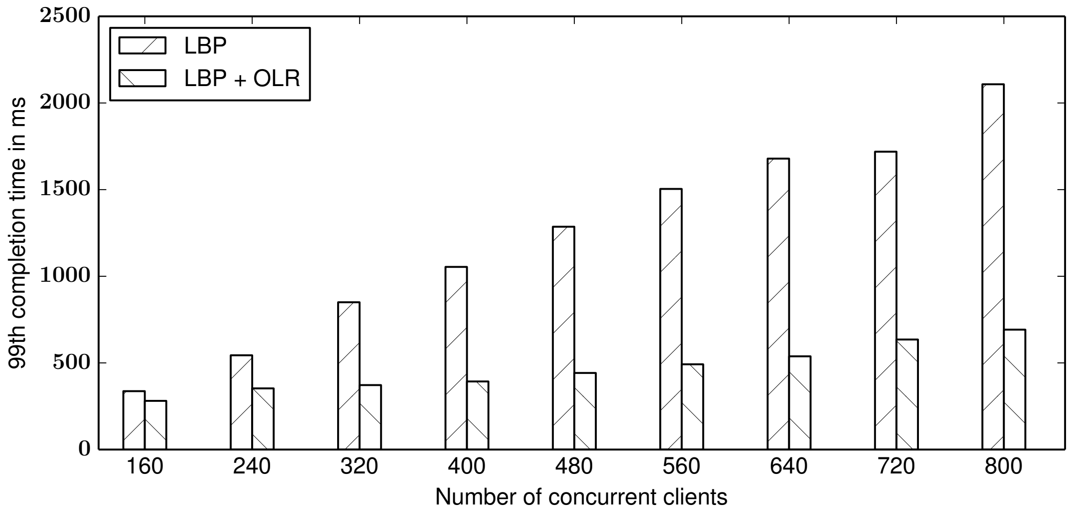

Figure 5.11: LBP vs (LBP + OLR) - 99th percentile interaction completion time for ViewUserInfo

As shown in Figure 5.10, OLR further reduces the 95th percentile completion times for the ViewUserInfo interactions. Similar to the results we have seen in the preceding section, this improvement also diminishes as the number of clients increases. However, OLR significant reduces the 99th percentile completion times for ViewUserInfo, as seen in Figure 5.11. It is because OLR can significantly reduce the total number of cross-pod flows, which in turn reduces the load on the core link layer. Moreover, since our network is heavily oversubscribed at the core link layer, the tail completion times are likely caused by the cross-pod flows in the interactions. For this reason, reducing the number of cross-pod flows as well as reducing the core link layer load can greatly help to reduce tail completion times. Thus, OLR has bigger impact on the 99th percentile results than the 95th percentile results for the ViewUserInfo interactions.

Similar to the results in the last section, OLR can also greatly help to reduce both the 95th and the 99th percentile completion times for the StoreBid interactions as shown in Figure 5.12 and Figure 5.13. This is due to the complexity of the StoreBid interactions which gives OLR more space for improvement. The BrowseCategories graphs are omitted in this section because they show similar results to the StoreBid graphs.

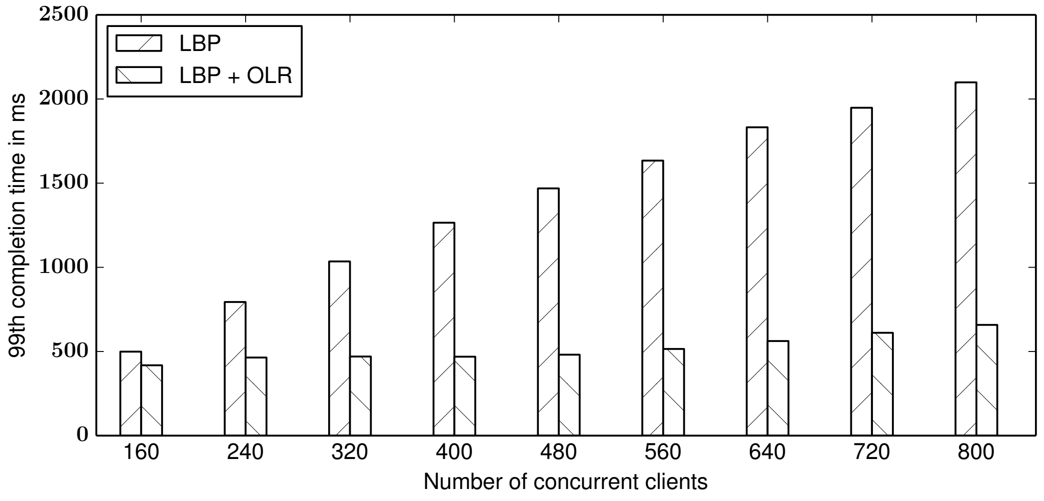

Figure 5.12: LBP vs (LBP + OLR) - 95th percentile interaction completion time for StoreBid.

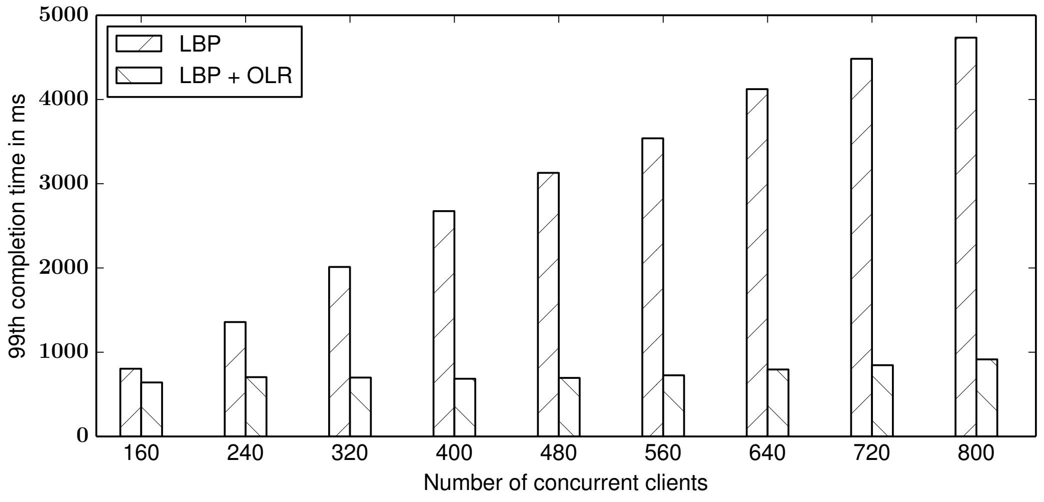

Figure 5.13: LBP vs (LBP + OLR) - 99th percentile interaction completion time for StoreBid.

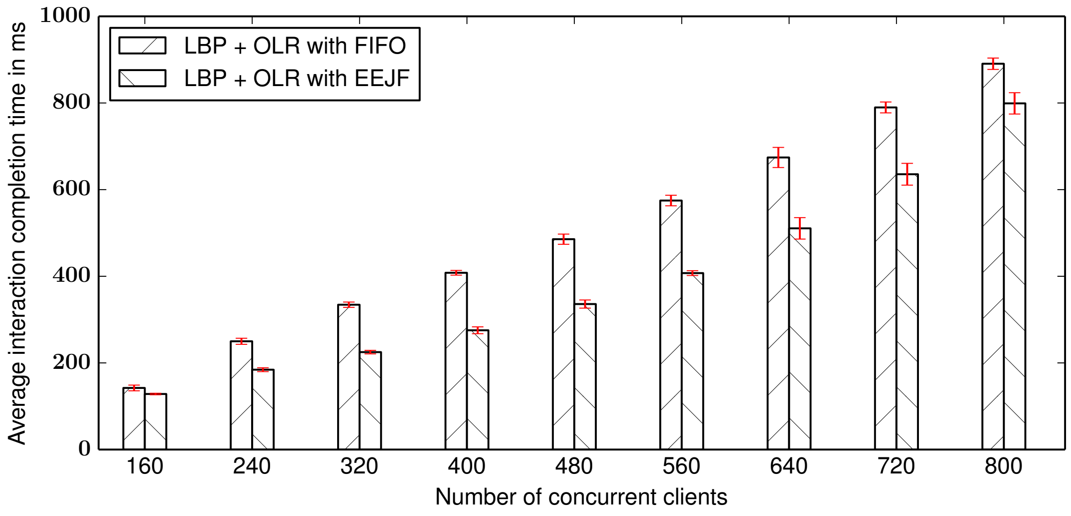

Figure 5.14: (LBP + OLR with FIFO) vs (LBP + OLR with EEJF) - Average interaction completion time.

5.4 Performance of OLR with Earliest Expected Job First

The third goal of the evaluation section is to show that the *Earliest Expected Job First* (EEJF) algorithm can further help OLR to reduce the load on the core link layer. In particular, we compare the default FIFO queueing of OLR with EEJF.

As shown in Figure 5.14, EEJF reduces the average interaction completion times by more than 30% for 320, 400, and 480 clients, while the improvement is smaller in other cases. The reason is that when the queue length is small, the impact of EEJF is likely to be small because there are not enough transactions in the queue for EEJF to reorder. However, when the queue length is large, OLR with FIFO already has many opportunities to redistribute the load on core link layer. Thus the impact of EEJF becomes relatively marginal. Moreover, this behaviour is also observed in the throughput graph in Figure 5.15. The bell-shaped curve of the throughput improvement shows that EEJF has the largest impact when the number of client is between 240 and 720.

Figure 5.16 and Figure 5.17 show that EEJF only improves the tail latency of the

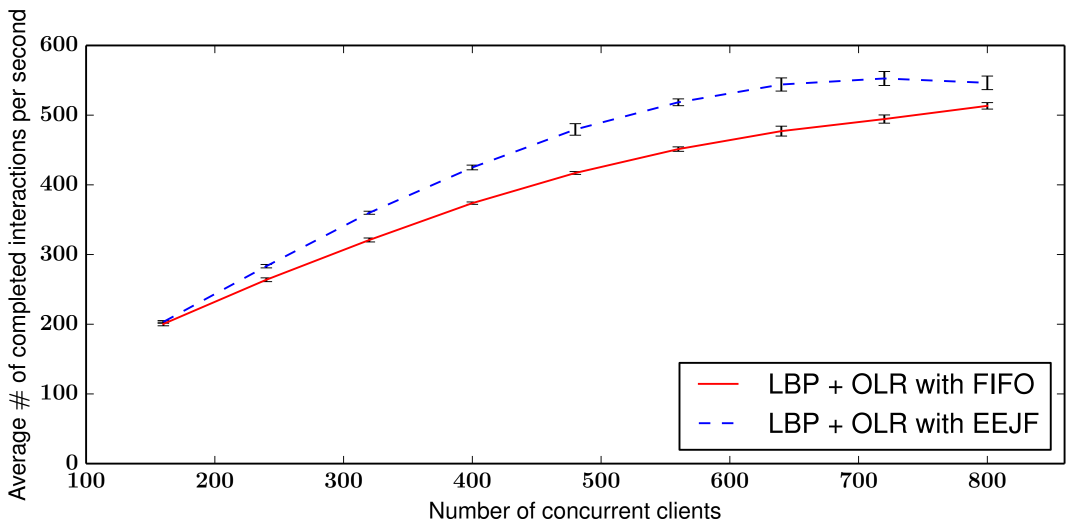

Figure 5.15: (LBP + OLR with FIFO) vs (LBP + OLR with EEJF) - Throughput.

ViewUserInfo interactions when there are 400 or fewer clients. We believe the reason is that as the pending queue length increases, the overhead of EEJF also increases, which in turn increases the tail latency of the ViewUserInfo interactions. The results for the StoreBid interactions are similar and therefore omitted. On the other hand, the percentile results for the BrowseCategories interactions show that EEJF can correctly identify the large flows which traverse the core link layer. As seen in Figure 5.18 and Figure 5.19, the tail latencies of BrowseCategories interactions is increased significantly when the system is heavily loaded. By delaying the large flows, which are likely to traverse the core link layer, OLR has a higher chance to redistribute the core link layer load, which in turn improves the system performance in terms of the average completion time and the throughput.

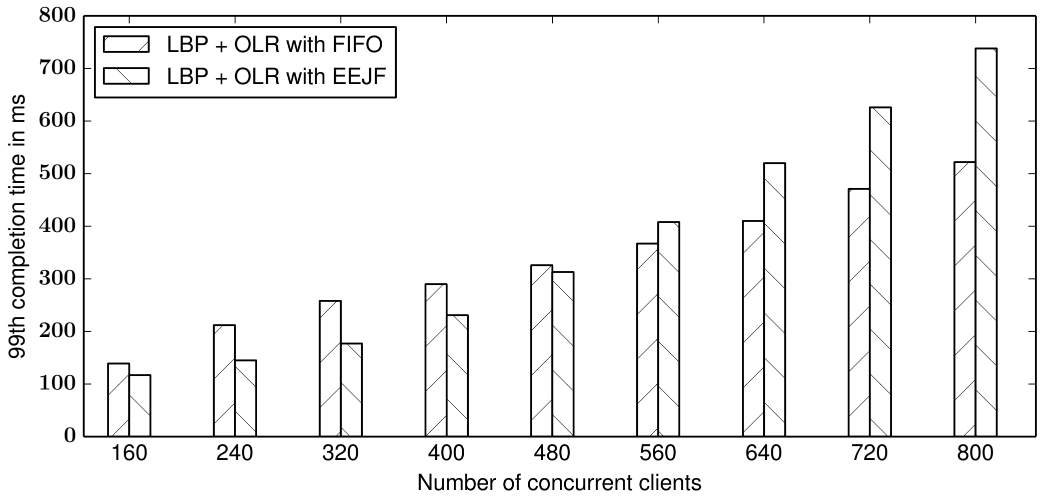

Figure 5.16: (LBP + OLR with FIFO) vs (LBP + OLR with EEJF) - 95th percentile interaction completion time for ViewUserInfo.

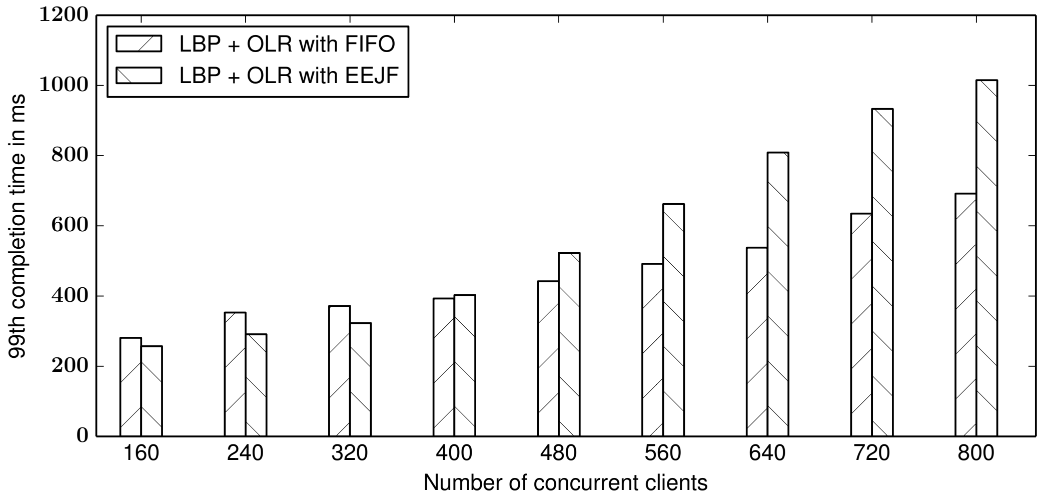

Figure 5.17: (LBP + OLR with FIFO) vs (LBP + OLR with EEJF) - 99th percentile interaction completion time for ViewUserInfo.

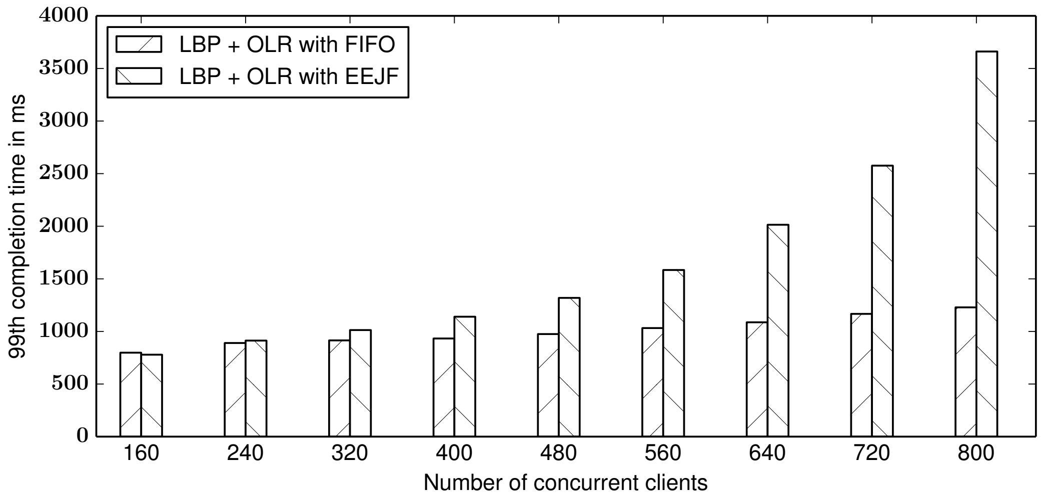

Figure 5.18: FIFO vs EEJF - 95th percentile interaction completion time for BrowseCategories.

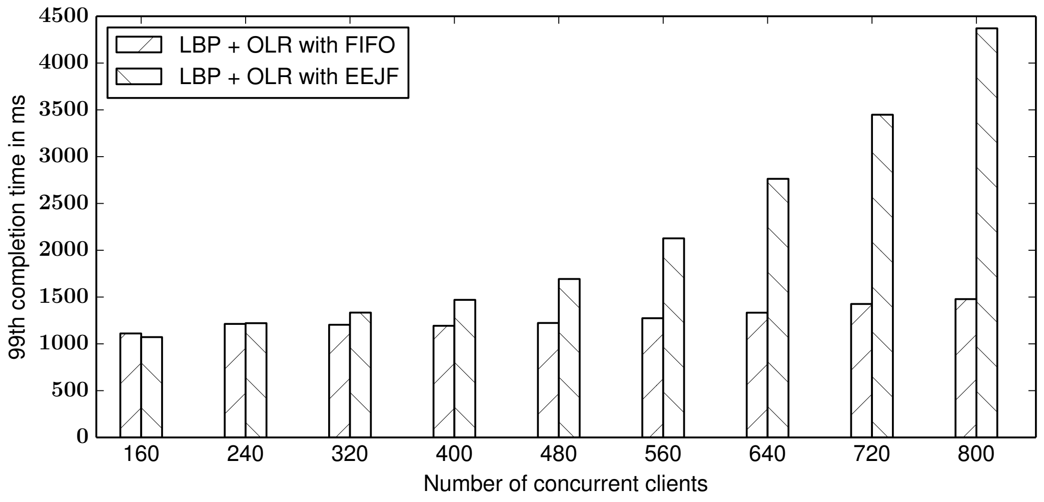

Figure 5.19: FIFO vs EEJF - 99th percentile interaction completion time for BrowseCategories.

FIFO	EEJF
45.0%	53.1%

Table 5.3: Replica read percentage with 400 clients.

EEJF	FIFO	FIFO with reduced probability of large flows
274.8	407.6	310.8

Table 5.4: Average interaction completion time of (OLR + LBP) at 400 clients.

To further understand the impact of EEJF on OLR, we have collected extra results from the experiments with 400 clients. We define the *replica read percentage* to be the number of replica reads, introduced by OLR, divided by the sum of replica reads and normal reads. Table 5.3 contains the replica read percentage for two interactions as a whole. Both of these interactions contain read operations on large data items in the RUBiS benchmark. In particular, the interactions are BrowseCategories and BrowseRegions. EEJF has increased the replica read percentage of the two interactions by 18.2%. We believe this is why EEJF is performing better than FIFO, because large flows on the core link layer have a larger impact on system performance. To validate our belief, we have performed one extra experiment to show the impact of BrowseCategories and BrowseRegions on system performance. In this experiment, we have reduced the probability of BrowseCategories and BrowseRegions by 20%. The result is shown in Table 5.4. We can see that the reduction of large flows have great impact on reducing the average interaction completion times. In particular, the average interaction completion time has decreased by 23.7% after reducing the probability of large flows.

Therefore, we conclude that EEJF helps OLR to reduce the core link layer load by accurately identifying and delaying large flows that traverse the core link layer, which in turn improves the system performance in terms of the average completion time and the throughput.

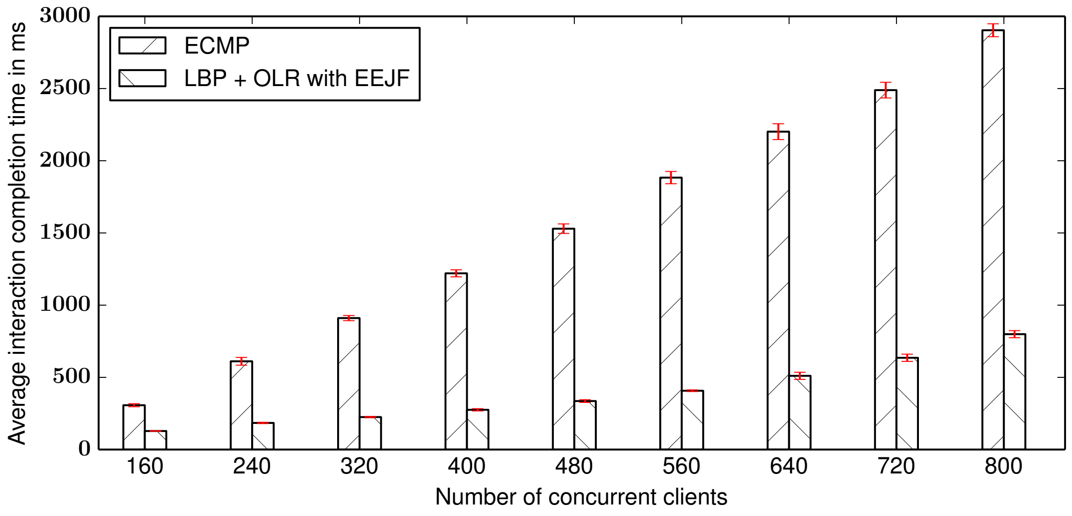

Figure 5.20: ECMP vs NetStore - Average interaction completion time.

5.5 NetStore Contribution to Performance

The last goal of the evaluation section is to illustrate the overall performance improvement that NetStore can provide. We compare all the techniques we have covered in this thesis with ECMP.

Figure 5.20 shows that NetStore has reduced the average interaction completion time by about 78% at 480 clients. Furthermore, NetStore has doubled the system throughput at 480 clients, as seen in Figure 5.21.

Last, Figures 5.22, 5.23, and 5.24 show that NetStore consistently outperforms ECMP for interaction tail latencies. In particular, NetStore has reduced the 99th percentile completion times by 64%, 75% and 63% for the ViewUserInfo, StoreBid, and BrowseCategories interactions at 480 clients.

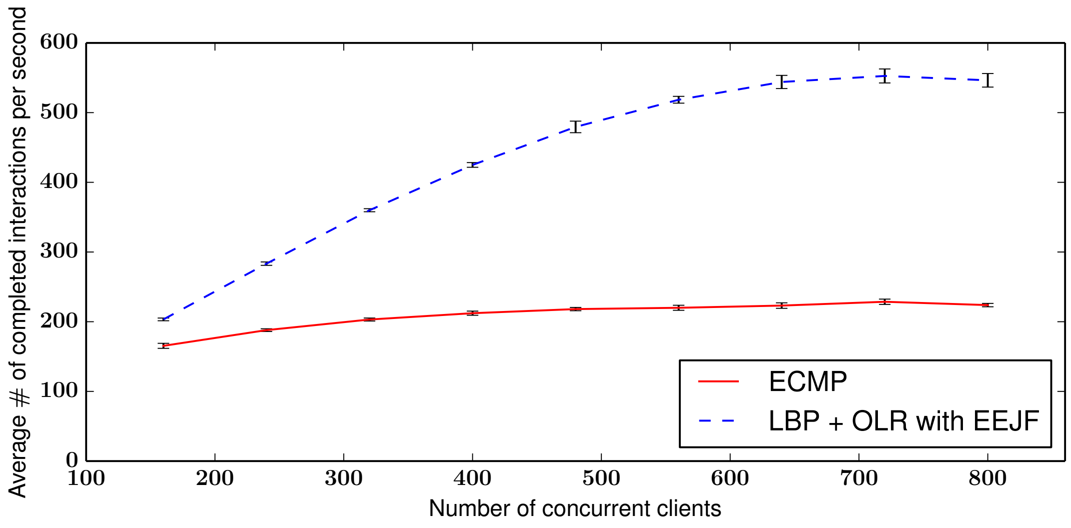

Figure 5.21: ECMP vs NetStore - Throughput

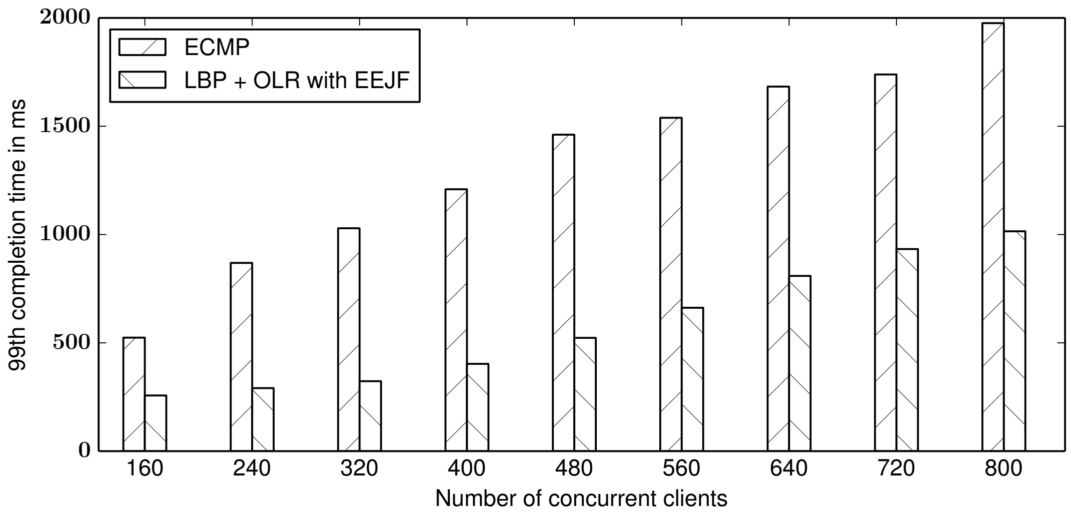

Figure 5.22: ECMP vs NetStore - 99th percentile interaction completion time for ViewUser-Info.

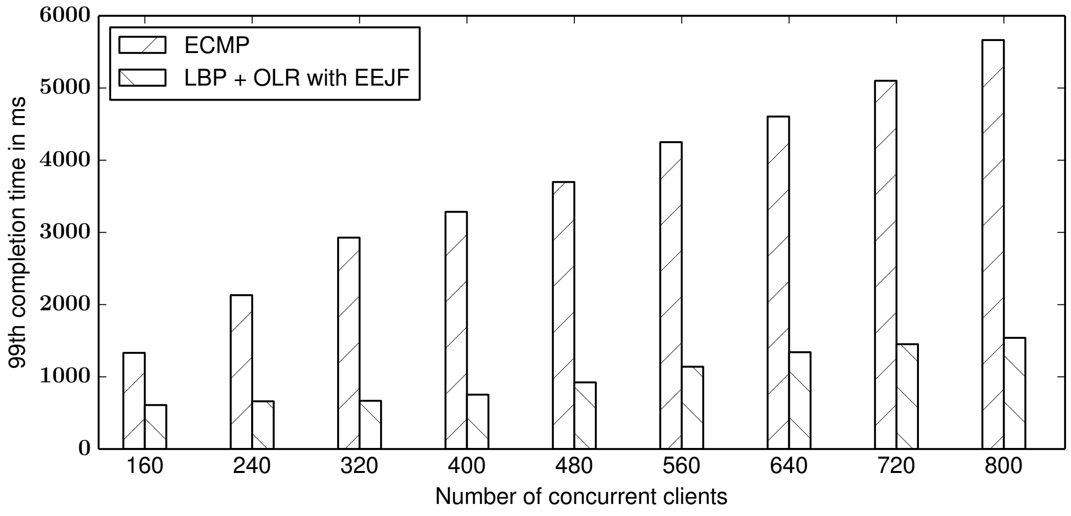

Figure 5.23: ECMP vs NetStore - 99th percentile interaction completion time for StoreBid.

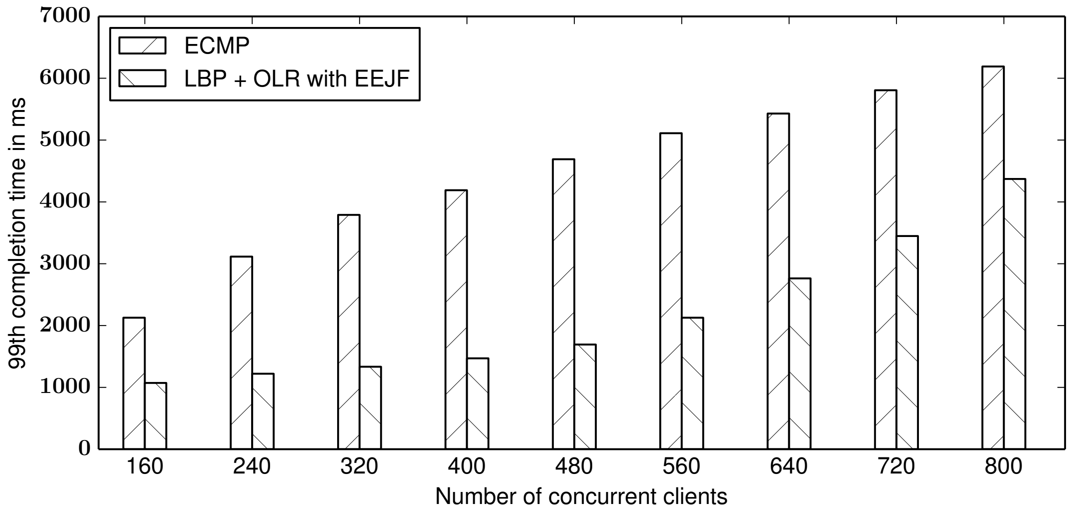

Figure 5.24: ECMP vs NetStore - 99th percentile interaction completion time for Browse-Categories.

Chapter 6

Conclusion

In this thesis, we have presented NetStore, a new distributed transaction processing system that bridges the gap between network research and distributed database research to avoid transaction performance deterioration due to network saturation. NetStore leverages the SDN technology with network layer and database layer optimizations to support transaction processing with network-awareness. In particular, NetStore incorporates the centralized lock server of traditional distributed database systems into the SDN controller. With the help of the SDN controller, NetStore is able to apply LBP, a network load balancing algorithm, at the network layer. Moreover, NetStore introduces OLR, a database layer load redistribution technique, to further improve the system performance. In addition, EEJF, a transaction scheduler, is introduced to assist OLR by building a performance model of the underlying system. Through experiments, we have shown that NetStore significantly improves the average transaction completion times, the transaction tail latencies, and the system throughput.

6.1 Future Work

There remains other optimizations that could improve the performance of NetStore and we note three interesting directions for future work in this section. First, NetStore currently does not employ any replication, but we feel this is an interesting area for the future as it provides more flexibility during scheduling. For instance, an operation may avoid traversing a congested link or the oversubscribed core link layer by fetching data from an appropriate replica. This will improve the performance of read operation at the cost of increased the response times for write operations. In addition, the replica placement problem will require further explorations. Second, we have discussed systems that assign response time deadlines to network flows to satisfy Service Level Agreement (SLA) requirements in Chapter 2. As an extension, NetStore can also assign a deadline to each transaction. For example, transactions that fetch data for a user-facing web page should have shorter deadlines than the transactions that fetch data for a business report. Last, NetStore currently focuses on improving the performance of short-lived flows that are dominant in transaction processing systems. We hope to extend our work to also optimize the performance of long-lived traffic. One possible solution is to recognize a flow as a long-lived flow after it has persisted for a predefined period of time and to treat long-lived flows differently using appropriate optimizations.

References

[1] Dscp. https://en.wikipedia.org/wiki/Differentiated_services.

[2] Floodlight openflow controller. http://www.projectfloodlight.org/floodlight/.

[3] Redis. http://redis.io/.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, New York, USA, 2007.

[5] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: dynamic flow scheduling for data center networks. In *NSDI'10 Proceedings of the 7th USENIX conference on Networked systems design and implementation*, Boston, USA, 2010.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *SIGCOMM '10 Proceedings of the ACM SIGCOMM 2010 conference*, New Delhi, India, 2010.

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM '13 Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, Hong Kong, China, 2013.

[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI'15 Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, Santa Clara, USA, 2015.

[9] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC '10 Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, New York, NY, USA, 2010.

[10] Theophilus Benson, Ashok Anandand Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40, 2010.

[11] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39, 2010.

[12] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 246–261, New York, NY, USA, 2002. ACM.

[13] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM '15 Proceedings of the 2015 ACM conference on SIGCOMM*, London, UK, 2015.

[14] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *SIGCOMM '14 Proceedings of the 2014 ACM conference on SIGCOMM*, Chicago, USA, 2014.

[15] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, September 2010.

[16] Phil Dixon. Site redesign: We get what we measure. Velocity Conference Talk, 2009.

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

[18] Robert Escriva, Bernard Wong, and Emin Gn Sirer. Hyperdex: a distributed, searchable key-value store. In *SIGCOMM '12 Proceedings of the ACM SIGCOMM 2012*, Helsinki, Finland, 2012.

[19] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peer-to-peer networks. In *IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems*, Cambridge, USA, 2002.

[20] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The architecture of pier: an internet-scale query processor. In *CIDR'05 Second Biennial Conference on Innovative Data Systems Research*, Asilomar, USA, 2005.

[21] Google Inc. leveldb: A fast and lightweight keyvalue database library by google. http://code.google.com/p/leveldb/.

[22] MongoDB Inc. mongodb. https://www.mongodb.org/.

[23] Namit Jain, Dhruba Borthakur, Raghotham Murthy, Zheng Shao, Suresh Antony, Ashish Thusoo, Joydeep Sen Sarma, and Hao Liu. Data warehousing and analytics infrastructure at facebook in proceedings. In *SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, IN, USA, 2010.

[24] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44, 2010.

[25] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Hotnets-IX Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Monterey, USA, 2010.

[26] Greg Lindem. Make data useful. http://glinden.blogspot.ca/2006/12/slides-from-my-talk-at-stanford.html, 2006.

[27] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38, 2008.

[28] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12, 2001.

[29] Nikos Ntarmos, Ioannis Patlakas, and Peter Triantafillou. Rank join queries in nosql databases. In *Proc. VLDB Endow.*, Hangzhou, China, 2014. VLDB Endowment.

[30] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *SOSP '13 Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Farmington, USA, 2013.

[31] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 592–603, March 2014.

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

[33] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[34] Akshay K. Signh, Xu Cui, Benjamin Cassel, Bernard Wong, and Khuzaima Daudjee. Microfuge: A middleware approach to providing performance isolation in cloud storage systems. In *2014 IEEE 34th International Conference onDistributed Computing Systems (ICDCS)*, Madrid, Spain, 2014.

[35] Kenn Slagter, Ching-Hsien Hsu, Yeh-Ching Chung, and Gangman Yi. Smartjoin: a network-aware multiway join for mapreduce. *Cluster Computing*, 17, 2014.

[36] Jonathan S. Turner and 1 David E. Taylor. Diversifying the internet. In *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, St. Louis, USA, 2005.

[37] Ricardo Vilaça, Rui Oliveira, and José Pereira. A correlation-aware data placement strategy for key-value stores. In Pascal Felber and Romain Rouvoy, editors, *Distributed Applications and Interoperable Systems*, volume 6723 of *Lecture Notes in Computer Science*, pages 214–227. Springer Berlin Heidelberg, 2011.

[38] Guohui Wang, T.S. Eugene Ng, and Anees Shaikh. Programming your network at runtime for big data applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 103–108, New York, NY, USA, 2012. ACM.

[39] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Scalable join queries in cloud data stores. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 547–555, Washington, DC, USA, 2012. IEEE Computer Society.

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

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