PAPER *Special Section on Data Engineering and Information Management*

Deeply Programmable Application Switch for Performance Improvement of KVS in Data Center

Satoshi ITO†, **Tomoaki KANAYA**†, ***Nonmembers***, **Akihiro NAKAO**††, **Masato OGUCHI**†††, ***Senior Members***, ***and*** **Saneyasu YAMAGUCHI**†a), ***Member***

SUMMARY The concepts of programmable switches and software-defined networking (SDN) give developers flexible and deep control over the behavior of switches. We expect these concepts to dramatically improve the functionality of switches. In this paper, we focus on the concept of Deeply Programmable Networks (DPN), where data planes are programmable, and application switches based on DPN. We then propose a method to improve the performance of a key-value store (KVS) through an application switch. First, we explain the DPN and application switches. The DPN is a network that makes not only control planes but also data planes programmable. An application switch is a switch that implements some functions of network applications, such as database management system (DBMS). Second, we propose a method to improve the performance of Cassandra, one of the most popular key-value based DBMS, by implementing a caching function in a switch in a dedicated network such as a data center. The proposed method is expected to be effective even though it is a simple and traditional way because it is in the data path and the center of the network application. Third, we implement a switch with the caching function, which monitors the accessed data described in packets (Ethernet frames) and dynamically replaces the cached data in the switch, and then show that the proposed caching switch can significantly improve the KVS transaction performance with this implementation. In the case of our evaluation, our method improved the KVS transaction throughput by up to 47%.

key words: *application switch, programmable switch, DPN, TCP migration, KVS, cassandra*

1. Introduction

In the computer network field, a network switch has been silent and transparent and not highly functional. Some functions, for example, ECN [1], RED [2], or CoDel [3], have been proposed for improving the functionality of network elements, but these do not allow network elements to work as complexly and functionally as a computer that has a large scale operating system. Recently emerging technologies of programmable control planes and data planes [4] are providing a novel network style, such as programming a network switch in order to improve the performance of an application running in a data center [5].

In this paper, we propose a concept of an application switch and a method for improving the performance of a key-value store (KVS) to address the challenge of improving application performance through a programmable network element. This paper is based on our previous conference papers [5]–[9]. The concept was proposed in a poster paper [6]. This concept focuses on an environment where the server administrator and the switch administrator are the same, and the administrator can run programs on the switch. In this concept, the administrator implements application functions, such as caching of KVS, on the switch to improve application performance. In the paper, we pointed it out that an application switch causes a compatibility problem with TCP. The papers [7]–[9] discussed the potential of the application switch with an experimental implementation of a cache-like function of KVS. In addition, we proposed a method to improve the performance of KVS constructed with Cassandra [10], [11] by implementing a caching function that monitors accesses and dynamically replaces the data in an application switch. We also showed a way to solve TCP's problems by adjusting TCP sequence numbers.

The remainder of this paper is organized as follows. Section 2 refers to related work. Section 3 explains our proposed concept of an application switch. Section 4 proposes a dynamic caching method using an application switch. Section 5 evaluates the proposed method. Section 6 presents a discussion, and Sect. 7 concludes this work.

2. Related Work

2.1 Software-Defined Network and Programmable Switch

SDN: McKeown et al. proposed OpenFlow [12] for SDN (Software-defined networking). This is one of the breakthroughs for flex programmable switches. Network elements in a network can be controlled in a logically centralized manner with OpenFlow. Only control planes can be flexibly controlled by software with OpenFlow, but this led to the concept of the programmable switch and was an innovative work.

P4: Bosshart et al. proposed the programming language P4 [4] for programming data forwarding planes. P4 can work with control protocols of SDN, e.g. OpenFlow. P4 enables the construction of new functions in a switch by not hardware implementation but programming. This provides a variety of benefits such as flexibility, ease to implement and test, cost reduction of implementation, and open architecture of

Manuscript received June 26, 2023.
Manuscript revised November 15, 2023.
Manuscript publicized January 17, 2024.

†The authors are with Kogakuin University, Tokyo, 163–8677 Japan.
††The author is with The University of Tokyo, Tokyo, 113–0033 Japan.
†††The author is with Ochanomizu University, Tokyo, 112–8610 Japan.
a) E-mail: sane@cc.kogakuin.ac.jp

DOI: 10.1587/transinf.2023DAP0009

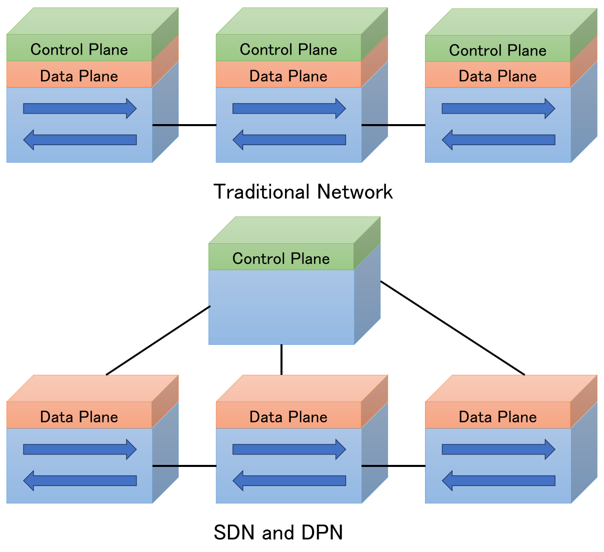


Fig. 1 Comparison of SDN, DPN and Traditional network

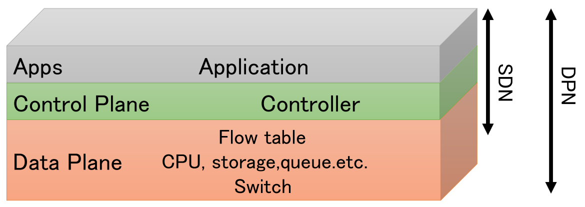


Fig. 2 SDN and DPN

network elements. Although P4 provides programming to monitor the data described in the header of each packet, it does not provide inspection of the packet payload, which is called Deep Packet Inspection (DPI). Exactly writing, a user can monitor the payload with P4 by defining their custom header in its payload. However, the user cannot define a variable-length header that requires lexical analysis.

DPN: Deeply Programmable Network (DPN) is a network and technology that enables data plane programming. FLARE is a network architecture for this network. A network switch is constructed using the Click Modular Router [13], which is a programming language for implementing a network element, on DPN. The language processor implementation of Click is open source software, and a developer can modify existing elements and create a new element of Click in the C++ language. A program can access and modify all bytes in a packet, including the payload, using this language. It can also support lexical analysis by implementing a Click element in the C++ language.

Figure 1 compares SDN and non-SDN networks. In the case of non-SDN, which is called the traditional network in the figure, each network element individually has a control plane, and each element must be controlled separately. In many cases, managing elements one by one is not easy and its cost is expected to be higher than the cost of managing all elements centrally. In the case of SDN including DPN, the control plane is logically consolidated. In this case, it is expected that many elements can be easily managed. Figure 2 illustrates SDN and DPN. In the case of SDN, an administrator can control not only the application and controller, but also the flow table of each data plane via software. In the case of DPN, an administrator can fully control and utilize even deeper devices such as its packet queue, the payloads of these packets, and files on the storage devices in the switch. These are easily controlled in a program written in C++.

2.2 TCP Migration

TCP migration is the process of changing one terminal, called a peer, of an established TCP connection to another terminal [14]. TCP identifies each connection with four numbers, which are the source and destination IP addresses and the source and destination port numbers. Therefore, these four values must be taken over to achieve transparent TCP migration without modifying the TCP protocol. In addition, when the connection returns to the old terminal, we call this *return migration* in this paper, the difference between old and new sequence numbers and Ack numbers should be managed appropriately. *Return migration* is described in Sect. 4.

2.3 Application Switch

In this paper, we propose an application switch in Sect. 3. The proposal is based on previous work as follows. First, we proposed to optimize the data forwarding plane in a switch based on DPI by programming the forwarding function [8]. Second, we proposed a switch with application functions inside [5]. In that work, our implementation only supported connectionless communication using UDP (User Datagram Protocol). Third, we pointed out that TCP connection migration and return migration are essential for accelerating TCP-based network applications [6]. Fourth, we focused on accelerating the performance of the database management system (DBMS) by implementing a caching function in an application switch [7]. In that work, we created a cache-like prototype system. The system cached the data that would be accessed again in the future, but it did not monitor the access and statically stored the fixed data without replacing the data in the cache. The application switch, not the server, responded to a query if the accessed data were in the cache. In addition, the prototype implementation did not support TCP sequence number management. We evaluated DBMS performance using the prototype implementation and showed its potential without considering the caching overhead and problems caused by TCP sequence number management, such as duplicate acks.

2.4 Cassandra Performance Improvement

A Cassandra system may consist of multiple server nodes. In this case, a client can send a query to any node in the system. When a client sends a query to a node, the node acts as a coordinator for that query. It acts as a proxy between the client and the nodes that have the requested data. It determines which nodes to forward the query to. Figure 21 illustrates this behavior. In the case of (a), the requested data is stored in Server2 and the client connects to Server1. Server1 acts as a coordinator and forwards the query to Server2. Namely, the query is transmitted as in A, B, C, and D. In the case of (b), the data is stored in Server1 and Client connects to Server1. In this case, Server1 replies the query without forwarding. The request is transmitted as in A and B. Obviously, in the case of (b), the client receives the response earlier than in (a).

Vakili et al. proposed a coordinator cache to improve the performance of Cassandra [15]. They proposed that the coordinator node has a caching function. If the data is stored in Server2 and the Client connects to Server1 multiple times, the coordinator cache in Server1 stores the data. After the coordinator cache stores the data as (c) in Fig. 21, Server1 responds to the request without forwarding the request. They then evaluated their method and showed that it could improve the performance of Cassandra.

However, they did not explain their method and implementation in detail while explaining the concept of their proposal. Therefore, their proposed method cannot be exactly implemented and replicated. Unlike our proposed method, their method caches the data not in the switch but in a server, their method has some disadvantages over our method. The query must be transmitted to a server node even in the case of a cache hit, while it is only forwarded to the switch with our method. In addition, the Cassandra process, which is a large process with high overhead, must be invoked even in the case of a cache hit. Furthermore, the method is not transparent and requires a user to modify the Cassandra implementation. The implementation cost of their method can be large with frequent updates of Cassandra implementation, while our method only needs to be updated when the specification of the query format is updated.

Gari et al. reported a detailed evaluation of the Cassandra performance on AWS (Amazon Web Services) [16]. They evaluated the performance in several scenarios in aspects such as the achieved KVS transaction throughput and CPU utilization. This comprehensive study is valuable and useful for using and improving Cassandra. We have proposed a method to improve Cassandra performance in an aspect of improving disk I/O [17]. We studied the file access frequency of Cassandra and found a large difference in access frequency per byte. Our method advises the operating system to keep some frequently accessed files in the page cache instead of managing the cached block based on LRU. We then showed that the method could improve the throughput of Cassandra transactions. However, the authors of these papers did not propose a method to improve performance in one aspect of network packet analysis.

2.5 Caching in Network Elements

In this subsection, we refer to work on caching in network elements. Barish et al. kindly reviewed the web caching technologies [18]. They introduced several proxy-based caching systems. Unlike our work, these are not transparent. Thus, they require manual or automatic configuration of the web browser. They also referred to and explained transparent caching methods. These methods do not require web browser configuration. Transparent caches work by intercepting HTTP requests in a network element, such as switches and routers, and redirecting them to web cache servers. For example, the Web Cache Communications Protocol (WCCP) [19] supports this transparent caching function based on packet analysis like our work. However, these transparent caching functions also have room for improvement. A program cannot be executed inside a network element. These methods only redirect requests to a cache server. As a result, unlike our approach, data cannot be cached in a switch at the center of a network. In other words, these methods do not exploit the potential of programmable switches. In addition, these methods are less advanced than our method. These methods do not analyze the response packets and do not extract data based on protocol analysis. Therefore, they cannot cache only data, such as key-value pairs, namely these methods process data in a unit of a file. We think this is mainly because these methods do not have enough programmability in a switch. This also causes the fact that these transparent caching functions cannot handle a write operation properly. There is another small disadvantage compared to our method. They are for web caching only. That is, many of these systems store the data based on the relationship between URL and data. These do not work well in many of personalized web site through cookies. We think one of the solutions for this problem is to handle data in finer grain size like our method. Liang et al. proposed a method for transparent distributed web caching [20]. This method also intercepts HTTP request in a network element and redirects the request to a web caching server. In the paper, the authors discuss in detail a load balancing method. However, similar to the work mentioned above like WCCP, the method also only redirects the request in a network element and a functional program cannot be executed in an element. Nirasawa et al. proposed a method for improving a testbed DBMS application by selecting the server to connect to based on the extraction of requested data from a packet [21]. This method slightly exploited the programmability of the switch, but did not fully exploit it. Their method does not create a reply packet or cache data in a switch. Therefore, data cannot be cached in a switch at the center of the network. Thus, we argue that a challenge and discussion on utilizing the programmability of switch such as our work is important over these existing works.

3. Application Switch

This section proposes an application switch.

3.1 Concept and Assumption

An application switch is a switch in which functions of a data center application, such as KVS, can be implemented

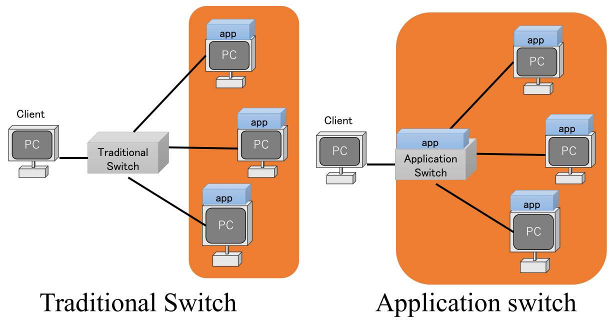


Fig. 3 Example of usage scenarios of normal and application switches

to improve its performance by flexibly controlling and optimizing the DPN [5] switch. The administrator can execute programs that can be implemented in Click or C++. Thus, a file in the file system in a switch can be easily read and written by the program. Figure 3 shows an example of assumed usage scenarios. In the case of a normal switch, a user places the user's application software on client and server computers. No application software is running on the switch. In the case of an application switch, a user places application software not only on these computers, but also on an application switch that is like a computer. In this scenario, the client computers, server computers, and an application switch are owned and managed by a single group. Then, the administrators of that group have administrative permissions to all of those computers, switches, and networks. They can place their application on them. The administrator lets the application on the switch inspect the packet payload. For example, a private cloud system in an enterprise and dedicated machines in a rack in a data center fit this situation.

Typically, a switch is placed at the center of a data center network, and many devices in the data center are directly connected to the switch. Therefore, we can expect that placing some important functions there can effectively improve application performance. However, this cannot be achieved with a traditional network switch, which only forwards frames according to the destination addresses in their frame headers. Programs run only on server and client computers. In the case of a DPN switch, a user-defined program can be placed and executed in a switch. For example, a developer of an application in a data center can implement a caching function in a switch that is in the data path of the application.

Based on these discussions, we propose an application switch that is not only a data forwarding plane, but also a platform for running a program. A developer can implement some functions of a data center application in an application switch in a programming language, mainly C++, using a DPN switch. As we described in Sect. 2.1, a program can read and write all the data in a packet (frame), which is a simple array in the C++ program. Thus, a program can analyze the packet payload and create a new packet in a switch.

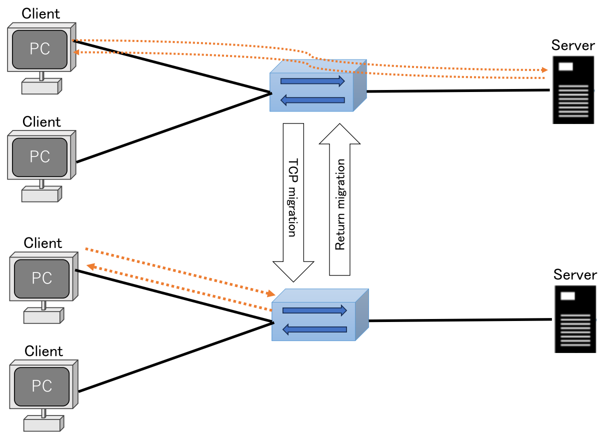


Fig. 4 The Overview of Application Switch

3.2 Application Switch for TCP-Based Application

This subsection proposes an application switch that supports TCP state tradition. This application switch migrates a TCP connection to improve TCP-based network applications.

Figure 4 shows the TCP connection *migration* and its *return migration* for server application performance improvement. In the case of a usual TCP and a switch that does not support TCP migration, a connection is established between client and server computers, and the peers of the connection do not change. A request from the client is always sent to the server and handled by the server. In the case of an application switch that supports TCP migration, the upper connection peer in the figure migrates from the server to the switch and migrates back to the server. In this paper, we refer to the former and latter cases as *migration* and *return migration*, respectively. To perform a migration, the application switch takes over the connection and receives the packet to the server instead of the server without forwarding it, as shown in the lower part of the figure. After *migration*, communication takes place between the client and the switch. To take over the connection, the switch must manage the TCP connection sequence number and Ack number. When a request from the client is replied by the switch, the request and reply are recognized only by the client and the switch, and not by the server. In other words, the sequence number and the Ack number in the connection increase from the client's and the switch's point of view, but they do not increase from the server's point of view. This makes a *difference* between the numbers managed by the client and the server. If the switch does not respond to a request in a packet, the switch forwards the packet as shown in the upper part of the figure. The migration from the lower situation to the upper situation is called *return migration*. After a *return migration*, the switch must modify the sequence number and the Ack number according to the *difference* between these numbers recognized by the client and the server. This modification is explained in detail in Sect. 4.2.

The application switch always inspects the payload of each packet. If it detects a request in the payload that the switch can reply, the switch does not forward it, but replies it. This causes *migration*. If it does not detect it, the switch forwards it to the server. This causes *return migration*.

4. Dynamic Caching by Application Switch

4.1 Caching Switch

In this section, we propose a caching system that is placed in an application switch in a network. It is for Cassandra. The cache in the application switch is initially empty. The switch continues to monitor packets and extracts the data access requests and their responses including the data. When it detects a request for non-cached data, it adds the data to its cache. If the cache is full, the newly accessed data replaces the least recently used (LRU) [22] data in the cache.

Figure 5 illustrates the behavior of the application switch with caching. If the target key-value pair of a query is not stored in the cache in the application switch (cache miss), the application switch simply forwards the query to the server. The server then returns the value to the client through the application switch. During forwarding, the application switch inspects the response packet, retrieves the value from the payload, and stores the key-value pair in its cache. If the target key-value pair is in the cache (cache hit), the switch does not forward the query, but creates the response and sends it to the client. Naturally, the cache results in a cache miss for almost all accesses immediately after starting with an empty cache, which is called a *cold cache*. After the cache space is filled with data, called the *hot cache*, accesses often result in cache hits and the system improves query processing performance.

There are two main ways to handle a write request in a caching environment, write-back and write-through. Cached data are stored redundantly in both the cache and the original memory. With write-through, the write request updates both the cached data and the original data. This is a pessimistic way. In the case of write-back, only the data in the cache are updated. This is an optimistic way. Our implementation currently supports only write-through for KVS. Thus, if the proposed caching system finds a write request, the system forwards the write request to the server, which is the original space, and updates the data in the cache. If write-back is supported, the caching system only updates the data in the cache when it finds a write request, and the original data in the server are not updated. This situation is called dirty, and the clients get the updated data from the cache. In many cases, the data in the original space, i.e. the server, are updated by multiple triggers. When the cached data is discarded from the cache, the updated data must be written back to the original space. In some implementations, such as the Linux kernel, the dirty cache is updated when the ratio of dirty data exceeds the threshold, when it has been dirty for longer than the threshold, and when a periodic flush event occurs. In some cases, cached data are purged when it has been cached for longer than the threshold.

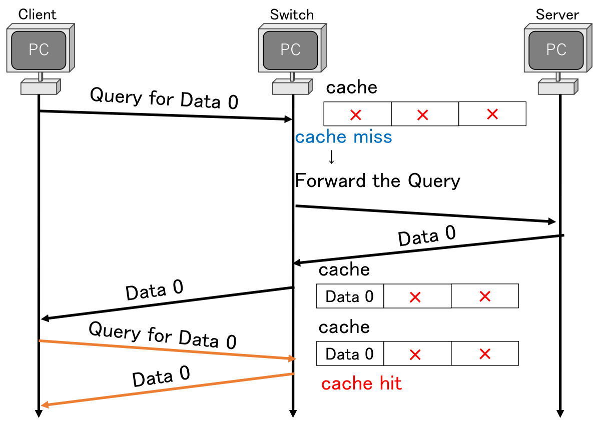


Fig. 5 Dynamic caching

4.2 Cassandra Query Analysis

In this subsection, we describe the analysis method, including the lexical analysis, of Cassandra queries in the proposed method. In a Cassandra query packet, the start of a field, such as the Key or Value field, stores its length, and the end of the field stores a vertical tab (0b in hexadecimal) or null character (00 in hexadecimal). Based on this protocol format, the proposed method analyzes the packet payload and retrieves the data between the start and end points to obtain the values in the Key and Value fields. By implementing such a lexical analysis function, the values in these fields can be retrieved at the application switch even if the protocol contains variable-length fields.

4.3 Sequence Number and Ack Number Management

The reply by a switch causes a sequence number mismatch between the client and the server. After a switch replies to a request, the request and its reply are known only to the client and the switch. That is, the server does not know them. Therefore, the sequence numbers managed by the client and the server will be different. The application switch must manage this difference after the *return migration* and correct the sequence number and Ack number.

First, for comparison, we illustrate the operation of a normal switch in Fig. 6. In this case, the packet arriving at this switch from the left, packet 0, is simply forwarded to the destination, and then packet 1 goes to the right. Each packet has a TCP header, and the header describes the sequence

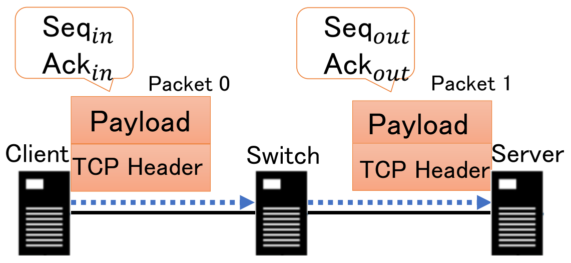


Fig. 6 Seq and Ack in normal switch

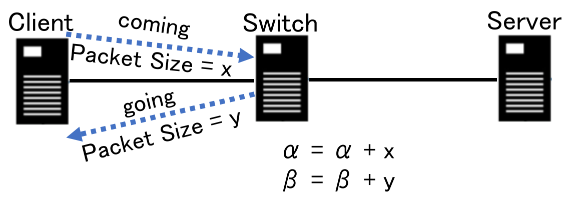


Fig. 7 Management of difference of Seq and Ack in application

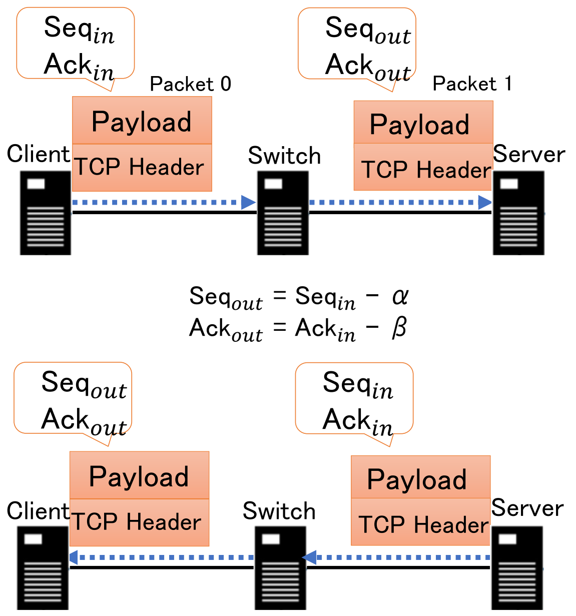


Fig. 8 Seq and Ack in application switch

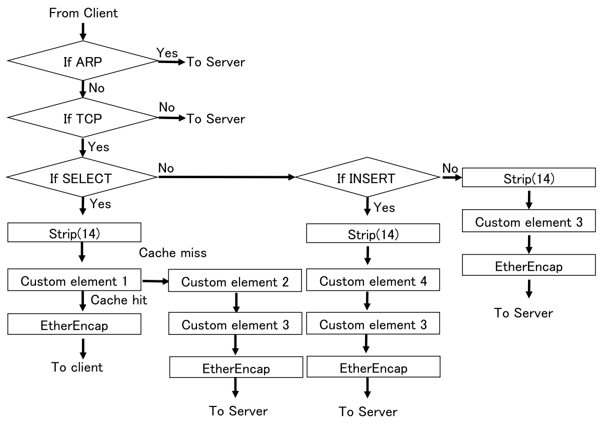


Fig. 9 Flowchart of application switch (packet from the client)

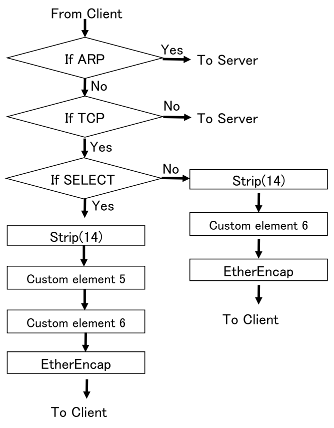


Fig. 10 Flowchart of application switch (packet from the server)

number and the ack number. Incoming and outgoing packets have the same sequence number. That is, $\mathrm{Seq}_{in} = \mathrm{Seq}_{out}$.

Second, we illustrate the operation of the application switch in Fig. 7. When a request packet arrives at the switch and the request results in a cache hit, the request packet is not forwarded to the server. The application switch creates a reply packet and sends it to the client as shown. In the figure, the sizes of the incoming and outgoing packets are x bytes and y bytes, respectively. The client's TCP observes the incoming packet, but the server's TCP does not. Therefore, the sequence numbers recognized by the client and the server differ by x bytes. The proposed switch manages this difference by α, in which case α is incremented by x. That is, $\alpha = \alpha + x$. Similarly, a reply packet whose size is y bytes is not observed by the server. The difference between the server and the client is maintained by β and it increases by y, so $\beta = \beta + y$.

An application switch modifies the sequence number and Ack number of a packet to be forwarded after these cache hits as shown in Fig. 8. For example, since the sequence number recognized by the server is smaller than the sequence number recognized by the client, an application switch reduces the sequence number and forwards this modified packet.

Here, we describe the problem caused by *duplicate Acks* [23] and the ack management policy of the proposed method. In Figs. 6, 7, and 8, a client sends a packet from left to right, and a server sends a packet from right to left. When a large number of cache hits occur consecutively, the same Ack packets with no payload and no window update are sent to the server many times, as shown in Fig. 5. These Ack packets are called *dubious* in the TCP implementation, and *duplicated Ack* indicates packet loss detection notification as a request for retransmission [24]. From the client side, if there are multiple of the same Ack, it looks like a *duplicated Ack*. To avoid this, in the proposed method, the same Ack number is transmitted only once. That is, if the Ack number has already been transmitted and the packet has no payload and no window update, the packet is not transmitted from the switch to the server. We explain how duplicate Acks occurs

4.4 Implementation

In this subsection, we explain the implementation of our application switch. It consisted of the Click source files and third-party click elements that we implemented. A Click source file is called a Click config file. It defines the flow and processing of incoming and outgoing packets by specifying the sequence of Click elements. The config files for processing packets from the client and server are shown in Figs. 9 and 10, respectively.

Next, we explain the elements within them. The *If SELECT* element analyzes the payload of each incoming packet and checks if it contains a data retrieval query (SELECT query). If it does, the element forwards it to the *Strip*(14) element. If it does not, the element forwards it to the next If element. The *If INSERT* and *If UPDATE* elements work similarly. *Strip*(14) is based on an existing Click element and strips the first 14 bytes from the packet (Ethernet frame) to remove the Ethernet header. In contrast, *EtherEncap* is an element to add an Ethernet header.

Custom elements in the figures are third-party elements that we have implemented. *Custom element 1* creates replay packets and is the most important element in our system. This element extracts the key specified in the payload and checks whether the key-value pair is stored in its cache. If the pair is in the cache, i.e. cache hit, this element generates a reply packet taking into account α and β. This element then sends this generated packet to the client via *EtherEncap* as shown in the path to below. If the pair does not exist, i.e. a cache miss, this packet is forwarded to the next element as shown in the path to the right.

In the case of a cache miss, *Custom element 2* receives the packet. This element maintains the Ack number in one direction for the server. This element does not send Ack packets with the same Ack number multiple times without a payload or window update to avoid unintentional duplicate Acks as described in Sect. 4.3.

Custom element 3 controls the difference in sequence numbers and Ack numbers caused by cache hits, and adjusts these numbers in packets in the direction from the client to the server based on the α and β of *Custom element 1*.

Custom elements 4 support Cassandra's *Insert* queries. This element parses incoming packets and extract the *Key* and *Value* described in the packets. They then update the cached data in the switch.

Custom element 5 is for processing reply packets from the server to the client. This element extracts the *Value* data from a packet and stores the data in the cache. If the key-value pair is in the cache, this updates the data. If not, it inserts the key-value pair.

Custom element 6 works similarly to *Custom element 3*. This element adjusts the sequence and Ack numbers in packets from the server to the client based on the α and β of *Custom element 1*.

We have not implemented an element to parse the UPDATE query. The INSERT query supports both adding a new key-value pair to a database and overwriting values in an existing key-value pair. Support for the UPDATE query is achieved by implementing an element to parse the UPDATE query according to its query format.

5. Evaluation

In this section, we evaluate the proposed method using a KVS system and a caching feature in an application switch. We have built a KVS system using Cassandra and the proposed application switch. The experimental network is shown in Fig. 11. The two clients and the server are directly connected to the application switch. The KVS implementation used is Cassandra 3.10. The specifications of the computers used in this experiment are described in Tables 1, 2, 3, 4, and 5.

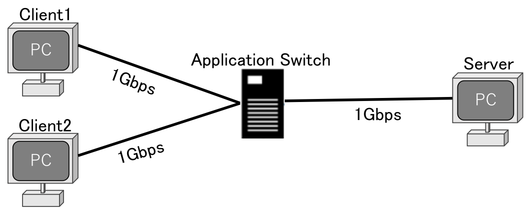


Fig. 11 Experimental Network

Table 1 Computer of server and server1

OS	CentOS 6.8
CPU	AMD Athlon(tm) II X2 220 Processor
memory	3.64[GB]
NIC1	Broadcom Tigon3 Gigabit Ethernet [Gbps]

Table 2 Computer of application switch

OS	CentOS 7.1 1503
CPU	Intel(R) Celeron(R) CPU 440 @ 2.00GHz
memory	1.01[GB]
NIC1 (connect to Client1)	Broadcom Tigon3 Gigabit Ethernet [Gbps]
NIC2 (connect to Client2)	Realtek RTL 8169 Gigabit Ethernet [Gbps]
NIC3 (connect to Server)	Realtek RTL 8169 Gigabit Ethernet [Gbps]

Table 3 Computer of client1 and client

OS	CentOS 6.5
CPU	AMD Athlon(tm) II X2 220 Processor
memory	1.77[GB]
NIC1	Broadcom Tigon3 Gigabit Ethernet [Gbps]

Table 4 Computer of client2 and server2

OS	CentOS 6.5
CPU	AMD Athlon(tm) II X2 220 Processor
memory	3.64[GB]
NIC1	Broadcom Tigon3 Gigabit Ethernet [Gbps]

Table 5 Computer of application switch2 and switch2

OS	CentOS 7
CPU	Intel Celeron CPU G530, 2.40GHz
memory	1.8 [GB]
NIC1 (connect to Client)	Realtek RTL8111/8168/8411 [Gbps]
NIC2 (connect to Server1)	Intel 82574L [Gbps]
NIC3 (connect to Server2)	Realtek RTL8111/8168/8411 [Gbps]

5.1 Query Processing Time

In this subsection, we compare the average turnaround times of reading queries with and without caching in the application switch.

We issued read queries to the KVS and measured turnaround times under the following conditions. We created

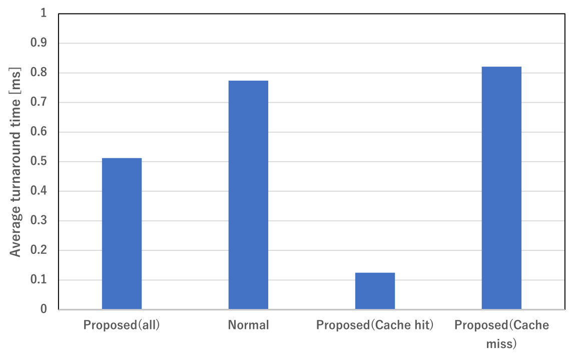


Fig. 12 Turnaround time of the normal and proposed methods

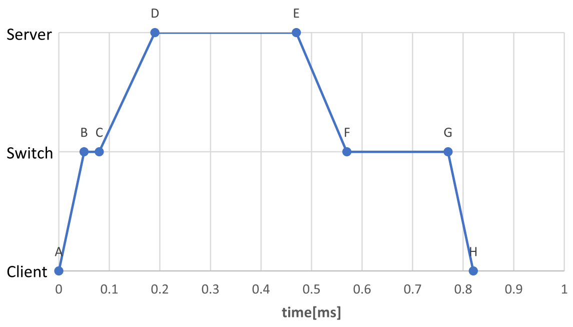


Fig. 13 Packet transmission over time (cache miss)

16 tables in Cassandra. The length of each table name was different, ranging from 1 to 16 characters. Each table has 256 key-value pairs, and all tables have total of 4096 entries. The length of a key is equal to the length of the table name, the content of a value is a random alphabetic string, and the length of a value is between 1 and $4N$, where N is the length of the table name. The turnaround time was measured by issuing 1024 read requests from the client to the server on a randomly selected key based on the Zipf distribution. The application switch cache size of the proposed method was 64.

The experimental results are shown in Fig. 12. "Proposed (all)" in the figure represents the average response time for all 1024 queries by the proposed method, i.e. with cache. "Proposed (cache hit)" and "Proposed (cache miss)" show the average response times for only the cache hit and cache miss queries, respectively. The cache hit ratio in this case was 33%. "Normal" shows the average response time using the normal method, i.e. without cache. The figure shows that the average response time of the proposed method is 34% shorter than that of the normal method. The main reasons for this performance improvement are that the switch, which was closer to the client, replied to the requests, thus eliminating the time to forward requests to the server in case of cache hits in the proposed method, and that it took less time for a lightweight switch process to replied to queries than it did for a huge Cassandra process holding a large amount of data.

Figures 13 and 14 illustrate packet transmission over time for cache miss and cache hit cases, respectively. These

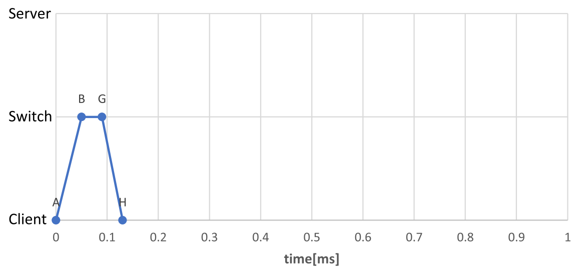


Fig. 14 Packet transmission over time (cache hit)

figures were generated by recording packet transmission and reception times on all network interfaces of the client machines, the application switch, and the server machine. The times obtained on these machines are not completely synchronized. We then adjusted the recorded time information to minimize the difference between the clock information of these machines. "A" in the figure is the time when Client1 transmitted the packet to the network. "B" is the time when Application Switch received (input) the packet from the network. "C" is the time when Application Switch transmitted the packet to the network. The time between "A" and "B" is the time spent to transfer the packet from Client1 to Application Switch, which is a propagation time between them. The time between "B" and "C" is the time spent for forwarding the packet by Application Switch. These figures indicate that the time taken by the application switch (from B to C) was less than the time taken by the server (from D to E). They also indicate that not sending packets to the server reduced the turnaround time.

Figure 12 shows that the average response time for a cache hit is much shorter (by about 84%) than for the normal method. On the other hand, the average response time for a cache miss is slightly longer (by about 6%) than that of the normal method. This is because the proposed method takes cache processing overhead and a miss penalty (e.g., the switch analyzes the response packet, extracts the value, and stores it in the cache) when a cache miss occurs, while the normal method only performs the frame forwarding function.

5.2 Cache Size and Performance

The relationship between cache size and response time was investigated. We varied the cache size from 4 to 1024 and measured the response time 1000 times under the same conditions as in Sect. 5.1. The experimental results are shown in Fig. 15. The bar and line graphs show the average response time and cache hit ratio, respectively. The figure shows that as the cache size increases, the cache hit ratio improves and the average response time decreases.

These results are supported by the results in Sect. 5.1. Namely, the proposed method decreases and increases the response time for cache hits and cache misses, respectively. Therefore, the higher the cache hit ratio, the better the performance.

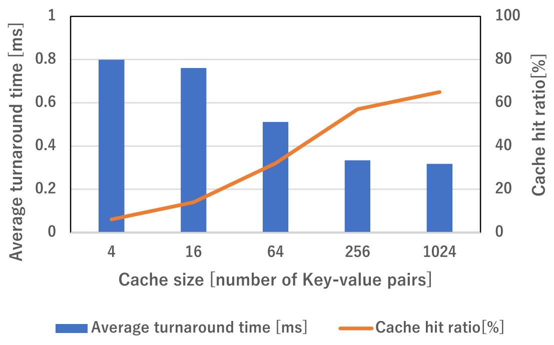


Fig. 15 Turnaround time and cache sizes

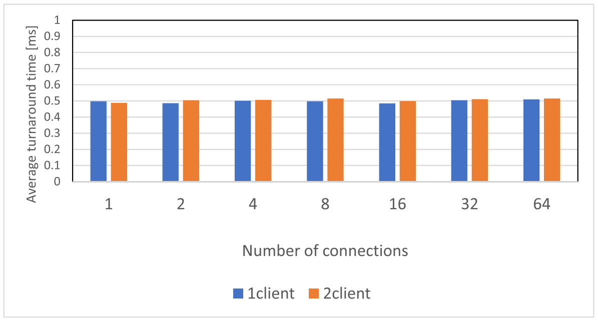


Fig. 17 Turnaround time, number of clients, and number of connections with accesses obeying the Zipf's law

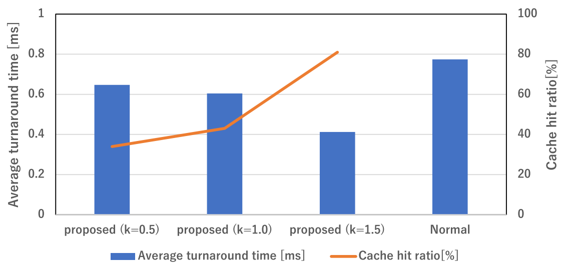


Fig. 16 Turnaround time with accesses obeying the Zipf's law

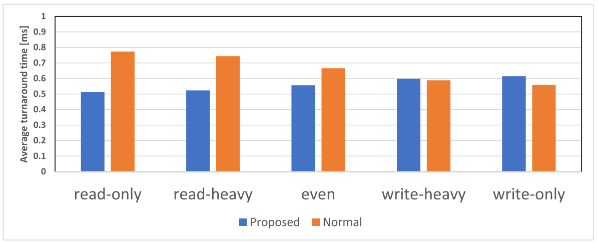


Fig. 18 Turnaround time and read/write ratio

5.3 Skew Strength and Performance

We then examine the relationship between the strength of the bias in the choice of query target and performance. Zipf's law can model many patterns of occurrence in practical applications. The model assumes that the probability of the n-th element is proportional to $1/n^k$. K is the parameter for controlling its bias. The larger k is, the stronger the bias. In many cases, k is one.

We varied k from 0.5 to 1.5 and evaluated the performance under the same conditions as in Sect. 5.1. Figure 16 shows the results. These show that the performance of the proposed method improved as the skewness of the access target distribution increased. Even in the case of the weakest skew with $k = 0.5$, the average response time of the proposed method was better than that of the normal method. In the case of the strongest skew with $k = 1.5$, the response time of the proposed method was 47% less than that of the normal method.

5.4 Performance and Number of Connections

In our method, all queries are routed through the application switch; if the application switch is overloaded, the switch may become a bottleneck and the performance of the network application (KVS in the case of this experiment) may be degraded. In this subsection, we examine the relationship between the load on the application switch and the response performance by varying the number of connections. Under the same conditions as in Sect. 6.1, the number of connections per client machine was changed to 1, 2, 4, 8, 16, 32, and 64, and the number of client machines was changed to 1 or 2, and the performance in each situation was evaluated.

The experimental results are shown in Fig. 17. The figure shows that there is no significant increase in the average response time as the number of connections and load on the application switch increases. This indicates that the application switch does not become a bottleneck and does not cause significant performance degradation in the load range of this experiment.

5.5 Read/Write Ratio and Performance

We investigate the relationship between the read/write ratio of queries and performance by varying the read/write ratio in 10:0 (read-only), 9:1 (read-heavy), 5:5 (even), 1:9 (write-heavy), and 0:10 (write-only). We conducted experiments under the same conditions as in Sect. 5.1. The SELECT and INSERT queries correspond to the read and write queries, respectively.

Figure 18 shows the performance. The red and blue bars represent the average response time of the proposed and normal methods, respectively. The figure shows that the proposed method improves the performance for read-intensive queries largely, but degrades the performance for write-intensive queries slightly. Compared to the advantage of the proposed method when the read ratio is high, the size of performance degradation of the proposed method when the read ratio is low is small. We then can expect caching by the proposed method to be effective in many situations.

5.6 Overhead of Lexical Analysis

Unlike programmable switches such as the P4 switch, the

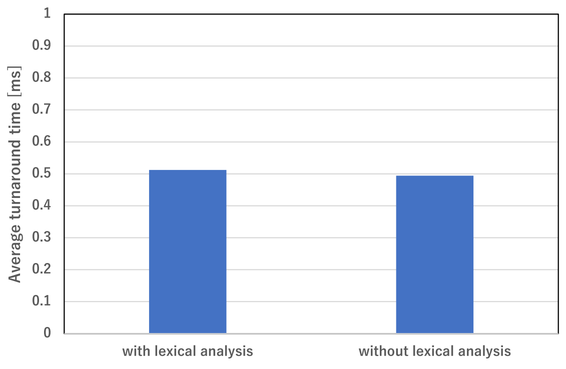


Fig. 19 Turnaround time of the proposed methods with and without lexical analysis

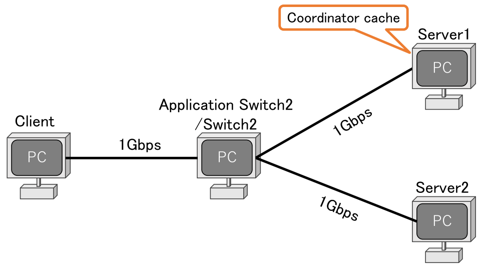


Fig. 20 Experimental Network II

proposed method performs lexical analysis in DPI and supports variable-length fields. In this subsection, we investigate the overhead of lexical analysis by comparing the performance of methods with and without lexical analysis. The experiments were performed with and without lexical analysis under the same conditions as in Sect. 5.1. The method without lexical analysis assumes that the field length is fixed and reads only the fixed address. The client then only sent queues that met the assumption.

Figure 19 shows the experimental results. The method with lexical analysis took a longer average response time than the method without analysis, but the difference is very small (about a 3% increase). The advantage of the caching function over the normal method is significant, and we argue that performing lexical analysis on Cassandra queries is well justified.

5.7 Performance with Multiple Servers

Here, we evaluated the proposed method with two server nodes. We have constructed an experimental network as shown in Fig. 20 and Tables 1, 2, 4, and 5 and evaluated the performance. The replication factor is one and the consistency level is ONE. In this environment, the existing method proposed by Vakili et al. described in Sect. 2.4 [15] can be applied. Therefore, we compare the performance of our proposed method and the existing method. The existing method is not explained in detail to be implemented, we have estimated the performance of the existing method as follows.

Client connects to Server1 and Server1 has a coordinator cache. First, we explain the cases of coordinator cache misses. In the cases, the data are replied by Server1 or Server2. If the requested data are stored in Server1, the data are replied by Server 1 like (b) in Fig. 21. Otherwise, the data are replied by the Server1 like (a) in Fig. 21. Second, we explain the cases of coordinator cache hit. In the cases, the coordinator cache in Server1 replies like (c) in Fig. 21. The behaviors of (b) and (c) are quite similar. Therefore, we estimate the performance of the existing method in the case of coordinator cache hit, i.e. (c), by the performance of coordinator cache misses and the data being stored in Server1, i.e. (a). Since the replication factor is 1 and the number of nodes is two, the data are stored in Server1 and Server2 with a probability of 50%.

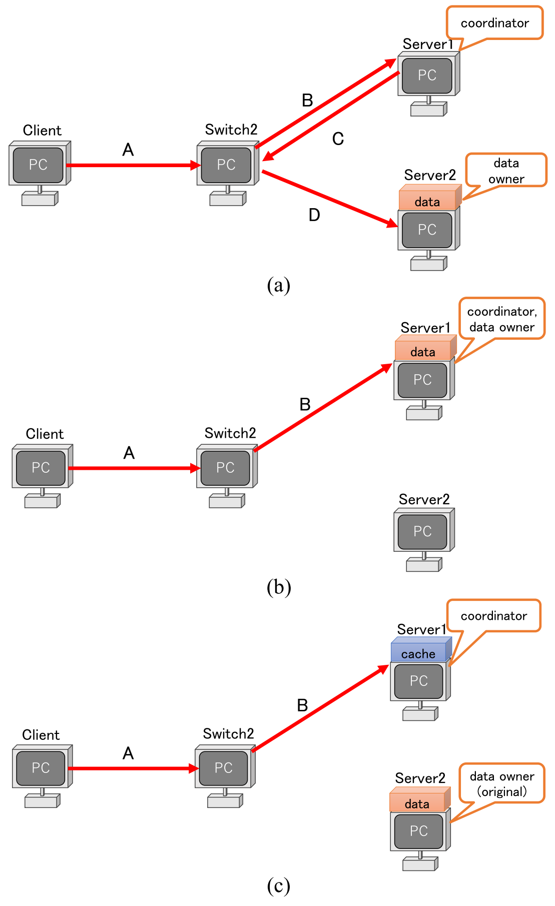


Fig. 21 Coordinator and Coordinator cache

Figure 22 depicts the measured experimental results and the estimated performances. The performances of the existing methods are estimated ones. The values of “Proposed” and “estimated Existing” are the average turnaround times of 100 queries of the proposed and existing methods, respectively. Those labeled “(cache hit)” and “(cache miss)” are the average turnaround times in cases of cache hit and cache miss in all queries, respectively. Those labeled “(Server1)” and “(Server2)” are those in cases where data are stored in Server1 and Server2, respectively. In these experiments, the cache hit ratio of the proposed method was 34%. Namely,

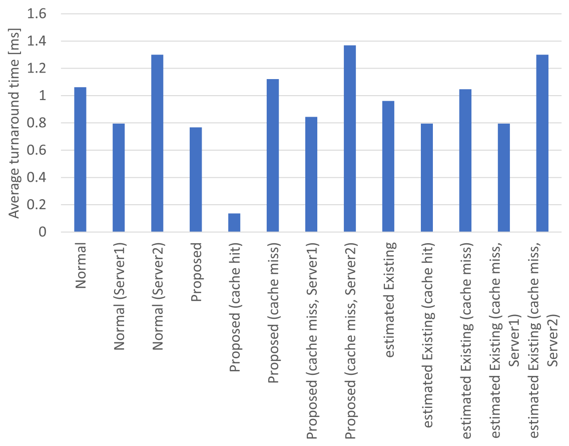


Fig. 22 Turnaround time with multiple servers

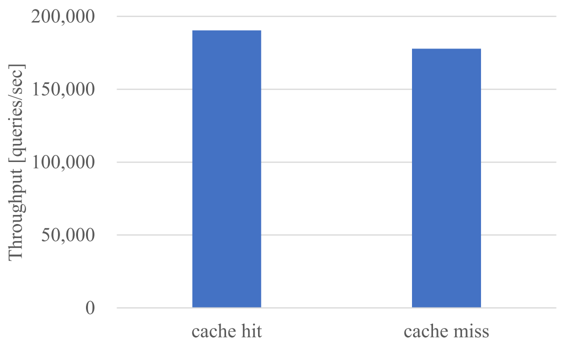


Fig. 23 Throughput (packets from a dumpfile)

the performance of the "*proposed*" is 0.34*"*proposed (cache hit)*" + 0.66*"*proposed (cache miss)*." The performance of "*proposed (cache miss)*" is 0.5*"*proposed (cache miss, Server1)*" + 0.5*"*proposed (cache miss, Server2)*." We estimated the performance of the existing method with the same cache hit ratio. Namely, the performance of "*estimated Existing*" is 0.34*"*estimated Existing (cache hit)*" + 0.66*"*estimated Existing (cache miss)*." "*Estimated Existing (cache miss)*" is 0.5*"*estimated Existing (cache miss, Server1)*" + 0.5*"*estimated Existing (cache miss, Server2)*." In this estimation, we do not include the overhead of caching process of the existing method. In practical implementation, the performance of the existing method may be slightly less than those performances. From these results, we can see that the turnaround time of our method in the cases of cache hit are remarkably less than those of the existing method. This is mainly because a query has to be transmitted to Server1 even if the cache hits in the existing method, the query is processed in the switch without forwarding in our method.

5.8 Estimated Performance Limitations

In this subsection, we estimate the performance limitation caused by an application switch. In the evaluation with the practical load in Sect. 5.4, we did not find performance limitation from the application switch. In this subsection, we highlight the performance limitation caused by an application. For this purpose, we evaluated the application switch throughput, i.e. number of processed packets per second, by inputting packets from a dump file to the application switch. In this case, packets can be provided to the application switch process within a quite short time. Namely, packets are provided with nearly infinite speed and only the processing performance of the application switch can be separately evaluated without many physical server computers. The specifications of the used computer is written in Table 5.

The experimental results are shown in Fig. 23. The application switches processed 10,000 packets in 52.5 ms in the case of cache hit. Namely, the application switch can process around 190,000 packets per second. It processed 10,000 packets in 56.2 ms in the case of cache miss, i.e. 178,000 packets per second. We can expect that the performances of an application switch will be limited by these values with a huge scale system.

6. Discussion

First, we discuss the future style of networking from the perspective of a programmable switch, DPI, and SDN.

TCP has mainly been used as a layer four protocol and will be widely used in the future, with a few exceptions such as QUIC [25]. These protocols, TCP and QUIC, assume that the connection is managed and recognized by both endpoints. Traditionally, with this assumption, network elements between these endpoints typically do not provide any information to the endpoints, even though these network elements have important information about the state of the network. In addition, a network element typically does not actively contribute to connections. As a result, TCP and QUIC implementations in endpoints must estimate network conditions based on some limited information, such as packet loss or network delay. There are a few exceptions, such as ECN and RED. A router with ECN enabled provides information. It sets the ECN bit in a packet and forwards it when the router is congested. The endpoints can read this bit for network status information. A router with RED enabled controls its queue with its predefined policy. The router will drop packets randomly in a situation where its queue length is large. However, the behavior of a router with these features is remarkably moderate, and the performance impact is very limited compared to DPN switches, SDN switches, and DPI switches.

Here we discuss two important networking styles in the future, the transparent style and the non-transparent style. The transparent style maintains compatibility with existing elements (hardware) and software. A traditional network switch is not intelligent, but only for forwarding frames to the destination described in the header. An application switch also should be compatible with existing elements and software, such as TCP. Namely, the switch is transparent and avoids errors in the TCP implementation by considering the state transition of TCP. In our research, the problems caused by TCP timeout [9] and duplicate acks [23] were pointed out, and we solved these problems by modifying the sequence numbers and ack numbers in the TCP headers. This transparent style will continue to play the most important role in the near future. However, the non-transparent style with highly functional switches, such as programmable switches, may change this situation. The latter style allows network elements to behave in complex and opaque ways. We believe that some small changes to the process of TCP will effectively improve this style of future network infrastructure supported by these technologies, i.e. SDN and programmable network. The problems caused by duplicate acks and sequence numbers will be easily solved by supporting a sequence number offset in the TCP option. When an application switch responds to a request with a cache hit, the application switch simply notifies this hit, i.e., an increase in the difference between the sequence numbers in the client and the server. After this notification, the client describes this difference in the offset field in the TCP option field. A server simply recalculates the sequence number and the ack number according to the offset written in the option. In this case, the difference is managed not by the switch in a network, but by the endpoints, i.e. the client and the server. This change drastically affects the concept of TCP. TCP guarantees that all bytes in the stream are transmitted correctly, but this option allows a server to skip some bytes of data. Although the impact on policy is not small, this option will help some functional behaviors in a switch with the reliability of TCP, such as retransmission and congestion control.

Next, we compare the consistency models of our proposed caching and the original Cassandra. We argue that our model is not far from the original Cassandra policy. Cassandra allows several consistency control policies, such as *write-all* + *read-one*, *write-one* + *read-all*, and *write-quorum* + *read-quorum*. For each policy, $R + W > N$ must be satisfied. R and W are the number of replicas covered by read and write operations, respectively. N is the number of replicas. Satisfying $R+W > N$ guarantees that a read operation after a write operation always accesses updated data. After updating W data, there are N-W unupdated data. Reading only these N-W unupdated data cannot satisfy $R+W > N$. Focusing on the key point that a read operation can access the latest data, we can consider that the proposed caching function satisfies this point. Fortunately, this caching assumes that all data access queries, including read and write, pass through the application switch. Therefore, the most recent data is always stored in the switch, and all read requests can get the most recent data. The coexistence of multiple caching functions on a network raises the issue of consistency between caches. In this paper, we have focused on the situation with a single caching function. Supporting multiple caching functions is an important future work.

Next, we discuss the pros and cons of parallelizing servers and using application switches to improve performance. Both are complementary. In other words, both can be used at the same time. The former is a conventional method that is effective at a cost. The latter is a method that has not been used before and is therefore expected to provide significant improvement, especially in environments with large network latencies. The former is a method that has been used and is expected to work without problems. The latter is a new method and may take some time to work properly. The latter is especially beneficial in a situation where it will not become a bottleneck. The former is very scalable.

Here, we discuss benefits of cache functionality implemented on the level of application switch by comparing to the implementation on server side. Server node usually has a caching function at DBMS level or operating system level. This caching function also improves the DBMS performance by avoiding access to its storage device such as HDD (hard disk drive) or SSD (solid state drive). However, a query needs to be transmitted to a server node as (a) and (b) in Fig. 21. Therefore, the size of its performance improvement is smaller than that of our proposed method where a query is transmitted only to the switch. In addition, the DBMS server process, which is large in many cases, must be accessed even if the cache is hit. This also implies that the size of the improvement of our method, which does not access a server process, is larger. We can find an advantage of our method also in focusing on the switch load. In the case of Cassandra, a query is often forwarded to the server node that has the data via a switch, as in (a) in Fig. 21. This requires the switch to forward the packet twice. The switch must also forward the response packet. On the contrary, in the case of our proposed method, the switch is only required to reply once. Of course, caching in the switch increases the load on the switch. Thus, caching in a switch has both positive and negative effects on the load of the switch. As described above, the cache in the switch is located at the center of the network. Therefore, important data are stored in the server cache. We think that even more important data should be stored in the switch cache.

We compare the performance of functional application switches and hardware switches. For the highest throughput, hardware switches may be appropriate. In this case, intelligent functionality is not available from such switches. On the other hand, in many cases, such as data center networks, common and commodity network elements are often used. Application switches and other software-based switches are sufficient to achieve this commodity-based performance. As a result, in these many cases, software-based switches can provide equivalent performance to a hardware-implemented switch. We think that comparing the performance of an application switch by enabling and disabling the caching feature is a valid way to evaluate the application switch when considering using an application switch for such situations.

Finally, we discuss Cassandra's consistency model and our caching implementation. Cassandra uses the eventual consistency model [26], [27], also called optimistic replication. If no new update requests are made to the data, all accesses to the data will eventually get the most recent value with this consistency. In the case of our implementation in this paper, which uses a single caching function, we can easily avoid violating this consistency model with common cache behaviors, including both write-through and write-back. Even with a write-back policy, the data in the switch should be updated first, and all clients should have access to the most recent data. With a write-through policy, the original data in the server is updated immediately. Both policies can maintain consistency.

7. Conclusion

In this paper, we focused on programmable switches and proposed the concept of an application switch. With this concept, developers can implement and execute some functions of a data center application in a deeply programmable network switch. We then proposed to implement a caching function of a Cassandra KVS in an application switch. The switch supported TCP connection migration by changing the sequence number and ack number. Our evaluation, including a single caching function, showed that the caching function reduced the average response time of the Cassandra KVS by up to 47%.

For future work, we plan to evaluate our method with a larger experiment with more client machines, implement a write-back policy, and discuss supporting multiple caching functions.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers 21K11854 and 21K11874.

References

[1] S. Floyd, "TCP and explicit congestion notification," SIGCOMM Comput. Commun. Rev., vol.24, no.5, pp.8–23, Oct. 1994. DOI: https://doi.org/10.1145/205511.205512
[2] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," IEEE/ACM Trans. Netw., vol.1, no.4, pp.397–413, Aug. 1993, DOI: 10.1109/90.251892.
[3] K. Nichols and V. Jacobson, "Controlling queue delay," Commun. ACM, vol.55, no.7, pp.42–50, July 2012. DOI: https://doi.org/10.1145/2209249.2209264
[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," SIGCOMM Comput. Commun. Rev., vol.44, no.3, pp.87–95, July 2014. DOI: http://dx.doi.org/10.1145/2656877.2656890
[5] S. Nirasawa, M. Hara, A. Nakao, M. Oguchi, S. Yamamoto, and S. Yamaguchi, "Network Application Performance Improvement with Deeply Programmable Switch," Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services (MOBIQUITOUS 2016). ACM, New York, NY, USA, pp.263–267, 2016. DOI: https://doi.org/10.1145/3004010.3004030
[6] S. Nirasawa, A. Nakao, S. Yamamoto, M. Hara, M. Oguchi, and S. Yamaguchi, "Application switch using DPN for improving TCP based data center Applications," Internatfional Workshop On IFIP/IEEE International Symposium on Integrated Network Management (AnNet 2017), 2017.
[7] T. Kanaya, A. Nakao, S. Yamamoto, H. Yamauchi, S. Nirasawa, M. Oguchi, and S. Yamaguchi, "Intelligent Application Switch Supporting TCP," 2018 IEEE 7th International Conference on Cloud Networking (CloudNet), Tokyo, pp.1–4, 2018. doi: 10.1109/CloudNet.2018.8549392
[8] S. Nirasawa, M. Hara, S. Yamaguchi, M. Oguchi, A. Nakao, and S. Yamamoto, "Application performance improvement with application aware DPN switches," 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), Kanazawa, Japan, pp.1–4, 2016. doi: 10.1109/APNOMS.2016.7737290
[9] T. Kanaya, A. Nakao, S. Yamamoto, M. Oguchi, S. Yamaguchi, "Intelligent Application Switch and Key-Value Store Accelerated by Dynamic Caching," Proceeding of 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC 2020), DBDM 2020: The 5th IEEE International Workshop on Distributed Big Data Management, pp.1318–1323, July 2020.
[10] Apache CASSANDRA,http://cassandra.apache.org/ (accessed on June 20, 2023).
[11] Cassandra Query Language, https://docs.datastax.com/en/cql/3.3/cql/cqlIntro.html (accessed on June 20, 2023).
[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner "OpenFlow: enabling innovation in campus networks," SIGCOMM Comput. Commun. Rev., vol.38, no.2, 69–7, April 20084. DOI: https://doi.org/10.1145/1355734.1355746
[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek, "The click modular router," ACM Trans. Comput. Syst., vol.18, no.3, pp.263–297, Aug. 2000. DOI: http://dx.doi.org/10.1145/354871.354874
[14] TCP Connection Migration, https://tools.ietf.org/html/draft-snoeren-tcp-migrate-00 (accessed on June 20, 2023).
[15] L. Azizi Vakili and N. Yazdani, "Speed up Cassandra read path by using Coordinator Cache," 2021 26th International Computer Conference, Computer Society of Iran (CSICC), Tehran, Iran, pp.1–5, 2021. DOI: 10.1109/CSICC52343.2021.9420593.
[16] Subba Reddy Gari, Avinash Kumar Reddy, Performance evaluation of Cassandra in AWS environment: An experiment, 2017.
[17] S. Yamaguchi and Y. Morimitsu, "Improving Dynamic Scaling Performance of Cassandra," IEICE Trans. Inf. & Syst., vol.E100-D, no.4, pp.682–692, 2017. DOI: 10.1587/transinf.2016DAP0009
[18] G. Barish and K. Obraczka, "World Wide Web caching: trends and techniques," IEEE Commun. Mag., vol.38, no.5, pp.178–184, May 2000. DOI: 10.1109/35.841844.
[19] D. McLaggan, "Web Cache Communication Protocol V2, Revision 1," Aug. 2, 2012. https://datatracker.ietf.org/doc/html/draft-mclaggan-wccp-v2rev1-00 (accessed on Nov. 15, 2023)
[20] Z. Liang, H. Hassanein, and P. Martin, "Transparent distributed Web caching," Proceedings LCN 2001. 26th Annual IEEE Conference on Local Computer Networks, Tampa, FL, USA, pp.225–233, 2001. doi: 10.1109/LCN.2001.990791.
[21] S. Nirasawa, M. Hara, S. Yamaguchi, M. Oguchi, A. Nakao, and S. Yamamoto, "Application performance improvement with application aware DPN switches," 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), Kanazawa, Japan, pp.1–4, 2016. doi: 10.1109/APNOMS.2016.7737290.
[22] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," SIGMOD Rec., vol.22, no.2, 297–306, June 1993. https://doi.org/10.1145/170036.170081
[23] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681, https://tools.ietf.org/html/rfc 5681 (accessed on June 20, 2023).
[24] W. Vogels, "Eventually consistent," Commun. ACM, vol.52, no.1, pp.40–44, Jan. 2009. DOI: https://doi.org/10.1145/1435417.1435432
[25] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," Proc. Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17). Association for Computing Machinery, pp.183–196, 2017. DOI: https://doi.org/10.1145/3098822.3098842
[26] W. Vogels, "Eventually Consistent: Building reliable distributed sys-

tems at a worldwide scale demands trade-offs?between consistency and availability.," Queue, vol.6, no.6, pp.14–19, Oct. 2008. DOI: https://doi.org/10.1145/1466443.1466448

[27] P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond: How can applications be built on eventually consistent infrastructure given no guarantee of safety?" Queue, vol.11, no.3, pp.20–32, March 2013. DOI: https://doi.org/10.1145/2460276.2462076

Appendix: Duplicate Acks

Here, we explain how duplicate Acks problem occurs when a large number of cache hits occur consecutively.

When a cache hit occurs, the sequence number recognized by the client is updated, while the number recognized by the server is not updated. When the client receives data from the switch, the client sends an acknowledge packet to the server. The acknowledge number is updated from the client's point of view and it sends the updated number. On the other hand, the number is not updated from the server's point of view, and the server receives an acknowledge for the non-updated number. If a cache hit occurs multiple times, the client sends an Ack packet for every reply packet by the switch, and every Ack packet is transmitted to the server (without solving the duplicate Acks problem). From the client's point of view, every Ack packet has different Ack number. However, from the server's point of view, every Ack packet has the same Ack number. These Ack packets with the same Ack number look duplicate Acks from server's point of view.

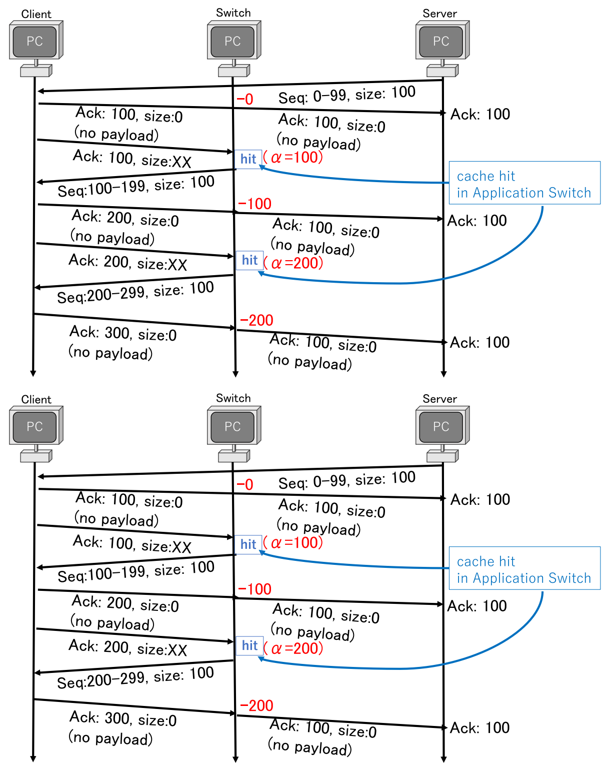
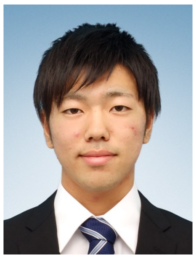


Fig. A·1 Duplicate Ack and Solution for it

Figure A·1 illustrates this behavior. The top half and bottom half represent behavior without and with resolution of the duplicate ack problem. In top upper half, the client sends three Ack packets. Each Ack packet has a different Ack number, namely Ack 100, 200, and 300. On the other hand, the server receives three Ack packets. All these packets have the same Ack number, which is Ack 100. The Ack packets by the clinet with 100, 200, and 300 are converted by the application switch into Ack packets for the server with 100, 100, and 100. In this case, the server receives the same number of Ack packets multiple times. As a result, the TCP implementation in the server detects duplicate Ack. As shown in the figure, each cache hit results in an Ack with the same number.

To solve this problem, the application switch does not forward an Ack packet with the same Ack number and no window update or payload.

Satoshi Ito received his B.E degree from Kogakuin University in 2022. He is currently a Master's student in Electrical Engineering and Electronics at the Graduate School, Kogakuin University.



Tomoaki Kanaya received his M.E degree from Kogakuin University in 2021.

Akihiro Nakao received B.S. (1991) in Physics, M.E. (1994) in Information Engineering from the University of Tokyo. He was at IBM Yamato Laboratory, Tokyo Research Laboratory, and IBM Texas Austin from 1994 till 2005. He received M.S. (2001) and Ph.D. (2005) in Computer Science from Princeton University. He taught as an associate professor (2005–2014) and as a professor (2014–2021) in Applied Computer Science, at Interfaculty Initiative in Information Studies, Graduate School of Interdisciplinary Information Studies, the University of Tokyo. He has served as Vice Dean of the University of Tokyo's Interfaculty Initiative in Information Studies (2019–2021). In April 2021, he has moved to School of Engineering, the University of Tokyo (2021-present). Since April 2023, he has been serving as Head of Department of System Innovations, School of Engineering. He was appointed as an adviser to the President of the University of Tokyo (2019–2020) and has been a special adviser to the President of the University of Tokyo (2020-present). He is serving as Director, Collaborative Research Institute for NGCI, (Next-Generation Cyber Infrastructure), the University of Tokyo (2021-present). For social services, he has been playing several important roles in Japanese government and also at research societies. He has also been appointed Chairman of the 5G Mobile Network Promotion Forum (5GMF) Network Architecture Committee by Japanese government. He has been appointed as Chairman of 5G/Beyond 5G committee, Space ICT Promotion Initiative Forum, International Committee, and Beyond 5G Promotion Consortium as well (2020-present). From 2020 to present, he is a chair and advisor of IEICE technical committee on network systems (NS) as well as a chair of IEICE technical committee on cross-field research association of super-intelligent networking (RISING). He will be the president of Communication Society, IEICE, in 2024.

Masato Oguchi received Ph.D from the University of Tokyo in 1995. He worked at NACSIS - currently known as National Institute of Informatics (NII), the Institute of Industrial Science in University of Tokyo, and the Research and Development Initiative in Chuo University. He joined Ochanomizu University in 2003 as an associate professor. Since 2006, he has been a professor at the Department of Information Sciences, Ochanomizu University. His research fields are network computing and data engineering, including high performance computing, mobile networking, network security, and information management. He is a member of IEEE, ACM, IEICE, IPSJ, and DBSJ.

Saneyasu Yamaguchi received his Ph.D. degree in Engineering from The University of Tokyo in 2002. From 2002 to 2006, he studied I/O processing at the Institute of Industrial Science, The University of Tokyo. He is now with Kogakuin University. His current research interests include operating systems, virtualized systems, and storage systems.