electronics

Article

Balancing Distributed Key-Value Stores with Efficient In-Network Redirecting

Yang Shi, Jiawei Fei, Mei Wen * and Chunyuan Zhang

National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China; shiyang14@nudt.edu.cn (Y.S.); feijiawei11@nudt.edu.cn (J.F.); cyzhang@nudt.edu.cn (C.Z.)
* Correspondence: meiwen@nudt.edu.cn

Received: 25 August 2019; Accepted: 6 September 2019; Published: 9 September 2019

Abstract: Today's cloud-based online services are underpinned by distributed key-value stores (KVSs). Keys and values are distributed across back-end servers in such scale-out systems. One primary real-life performance bottleneck occurs when storage servers suffer from load imbalance under skewed workloads. In this paper, we present KVSwitch, a centralized self-managing load balancer that leverages the power and flexibility of emerging programmable switches. The balance is achieved by dynamically predicting the hot items and by creating replication strategies according to KVS loading. To overcome the challenges in realizing KVSwitch given the limitations of the switch hardware, we decompose KVSwitch's functions and carefully design them for the heterogeneous processors inside the switch. We prototype KVSwitch in a Tofino switch. Experimental results show that our solution can effectively keep the KVS servers balanced even under highly skewed workloads. Furthermore, KVSwitch only replicates 70% of hot items and consumes 9.88% of server memory rather than simply replicating all hot items to each server.

Keywords: load balancing; key-value stores; programmable switches; replication

1. Introduction

Today's Internet services, such as search engines, e-commerce, and social networking, critically depend on high-performance key-value stores (KVSs). These KVSs must provide high throughput in order to process copious queries from millions of users while also meeting online response time requirements. In order to sustain performance above objectives, system operators increasingly rely on *in-memory* KVS deployed across multiple servers [1–3] using techniques like consistent hashing [4].

Although sharing data using multiple servers enables massive parallelism, such a scaling KVS, whether *in-memory* or not, can suffer due to hot items, namely those items that receive far more queries than others. The set of hot items changes rapidly due to popular posts and trending events [5–7]. This is because KVS workloads typically exhibit highly skewed request patterns. For example, Facebook reports that, in their KVS cluster, 10% of items account for around 60–90% of all queries [1]. In the presence of this skew, the servers holding the hot items become saturated, causing performance degradation (throughput is reduced, and response time increases).

A number of techniques have been proposed to mitigate this well-established skew problem. These techniques can be broadly divided into two classes: the first class [8–10] uses a dedicated cache to store hot items and to absorb the associated queries, while the second [3,11,12] replicates hot items across multiple servers and redirects queries to these servers to avoid imbalance.

Both categories of techniques have disadvantages or limitations. Cache-based techniques are mainly limited by the ability of the cache node. Since hot item queries account for a large proportion of real-life workloads, a single cache node is unlikely to keep up with the load. The second class of techniques requires mechanisms to redirect queries to different servers, typically using software load balancers (SLBs); however, an SLB can only process about 10 million queries per second [12], so a large number of SLBs are needed to balance the load of a distributed KVS.

To overcome these limitations, in this paper, we present KVSwitch, a switch-based load balancer for KVS that intelligently replicates hot items. Unlike previous replication-based techniques, KVSwitch does not require lots of SLBs. Instead, it exploits the power and flexibility of new-generation programmable switches to automatically redirect queries to the appropriate servers.

Realizing a load balancer for KVS in a programmable switch is challenging, as it needs not only to redirect KVS queries at line speed but also to dynamically identify hot items and to manage the replicas according to the workload. Inside the switch, two types of programmable processors are available: an application-specific integrated circuit (ASIC) pipeline and a standard CPU [13]. Neither processor can support the work of KVSwitch by itself. The ASIC pipeline does not support the complex computation requested in KVSwitch, while the CPU is limited by its throughput to support per-packet operation.

Rather than trying to shoehorn the KVSwitch into either the ASIC pipeline or CPU, we instead decompose it into two parts that are well suited for individual processors. The packet processing pipeline is designed to perform per-packet operations, including redirecting queries for hot items and recording of workload statistics. We carefully allocate and organize match-action tables, registers, and metadata to realize the operations mentioned above within the capacity limit of hardware resources. Meanwhile, the prediction and replication of hot items is performed on the switch CPU, since these operations require complex computations. Each processor relies on the other to overcome its limitations; together, they constitute KVSwitch.

Another challenge for KVSwitch is that of maintaining consistency between replicas, as any replica that is not up-to-date is invalid. Considering that KVSwitch lies in the ToR (Top of Rack) switch, one advantage of this is automatic awareness of all KVS packets targeting the rack, which includes information about replicas. Consequently, we design a multicast mechanism in the ASIC pipeline to spread the packets updating hot items to all replicas.

To summarize, our main contributions are as follows:

- We design the KVSwitch, an in-network balancer that uses programmable switches to provide dynamic load balancing for KVS without the need for additional hardware (Section 3).
- We devise a replication strategy that dynamically identifies hot items from the changing workload, which can significantly reduce unnecessary replicas (Section 4.1).
- We design a packet processing pipeline that efficiently detects and redirects queries for hot items and accurately collects the workload statistics (Section 4.2).
- We prototype KVSwitch in a Tofino switch and perform evaluations on it. The results demonstrate that KVSwitch is able to achieve satisfactory load balance with only 9.88% resource consumption on servers compared to simply copying hot items to all servers (Section 5).

This article is an extended version of the paper in Reference [14] presented at the Twenty-Fourth IEEE Symposium on Computers and Communications (ISCC 2019). Compared to the original conference paper, this one shows an exhaustive comparison with existing solutions and more details in the proposal, including more evaluation analyses to validate the proposal. Moreover, simulations are conducted to verify the proposal's scalability.

2. Motivation

2.1. Skew and Load Imbalance of KVS

Earlier research analyzing data access patterns in real-world settings have shown that the popularity of individual items in a dataset often follows a power–law distribution [6,7,15,16]. In such a distribution, a small set of popular items are queried far more often than other, "cold" items, leading to severe load imbalance across storage servers. This skew can be accurately represented by a Zipf distribution, in which the popularity of an item is inversely proportional to its rank. Zipf has a skewness parameter β, a higher value of which denotes a more acute imbalance. The most common value for β in the literature is 0.99 [5,8,9,11]; this value is also used in evaluations for KVSwitch (Section 5).

Skewness in workload results in load imbalance across storage servers. With $\beta = 0.99$ for 10,000 items, the hottest 100 items account for about 51.8% of all queries and the busiest server receives $7\times$ the average load [11]. As shown in Figure 1a, the server responsible for hot item Z experiences far greater loads than the server containing cold item D. A similar problem is also observed in the industry; in KVS for Alibaba, the world's largest retailer, a few servers will crash under the weight of numerous queries while many other servers are almost idle with no mitigation techniques [3].

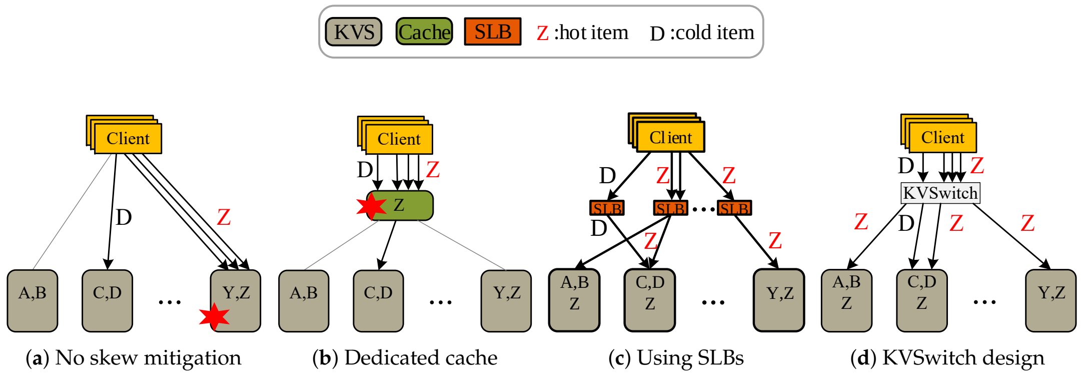


Figure 1. Design of different skew mitigation techniques.

2.2. Existing Solutions for Skew Mitigation

Figure 1b,c depicts two widely used approaches to skew mitigation: caching and replication.

Caching: The basic idea of cache-based approaches involves the use of a dedicated cache server to absorb hot item queries. In Figure 1b, all queries for hot item Z are processed by the dedicated cache. Since a small set of hot items are responsible for most queries, the skew-filtered queries should be balanced for back-end servers. These caching approaches suffer from two key limitations. First, these techniques usually target traditional flash-based or disk-based KVSs, since they are much slower than in-memory caches. Thus, a dedicated node with an in-memory cache is sufficient to keep up with the hot item load. However, this method does not work for in-memory KVS, as there is little difference in performance between the back-end servers and the cache node. Secondly, caching approaches are not viable at larger scales. In these situations, the load of hot items increases significantly, such that the caching method is fundamentally limited by the ability of the cache server.

Replication: As shown in Figure 1c, it is natural to copy hot items to multiple servers so as to achieve load balance. Accordingly, Z is replicated to another two servers so that the queries for it can be dispersed across these servers. Existing replication approaches typically rely on using SLBs to realize the redirect process for hot item queries [1,12]. However, lots of SLBs are required due to its low processing rate; updating replica information for larger numbers of SLBs is also complex. In order to simplify the updating process, Alibaba's KVS chooses to replicate all hot items to a hot zone on all servers. The price of this is the production of too many replicas in a cluster, increasing server memory usage significantly.

2.3. Programmable Switch

The recently proposed Reconfigurable Match-action Table (RMT) architecture [17], along with its upcoming products like the Barefoot Networks' Tofino switch chip, makes switches much more powerful and flexible. Instead of implementing specific network protocols, RMT designs an ASIC pipeline consisting of match-action stages. This ASIC pipeline has been deeply optimized to support packet processing at a line rate and to allow users to program packet behavior. In general, programmable switches provide users with both high throughput and great usability.

More specifically, the ASIC pipeline can be programmed in three aspects: (i) the switch parser can be programmed to identify custom packet formats; (ii) the match-action stage can be programmed to perform custom matching and actions; (iii) the on-chip memory can be programmed to store custom stages of packets. In this paper, we try to leverage programmable switches to make an in-network load balancer for KVS a reality.

3. KVSwitch Overview

KVSwitch is specifically designed for the ToR switch to provide load balance across storage servers inside this rack. This rack is dedicated to key-value storage, and all key-value items are partitioned to different servers based on a consistent hash. The proposed solution is presented in Figure 1d. KVSwitch achieves load balance by redirecting the KVS queries to the appropriate servers.

3.1. Network Protocol

Figure 2 shows the packet format of KVSwitch. To facilitate cooperation with existing protocols, the packet header of KVSwitch is inserted into the L4 payload as an application layer protocol. We choose user datagram protocol (UDP) for KVSwitch read queries and transmission control protocol (TCP) for write queries, which is consistent with KVS in the industry [1]. A special port is reserved for KVS packets so that KVSwitch can distinguish them from normal packets. KVSwitch is invoked in the switch only when this special port is detected.

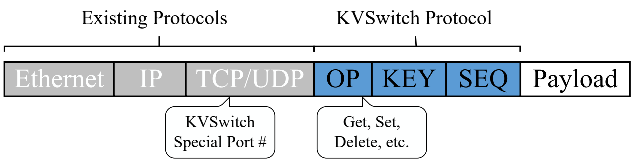


Figure 2. Network protocol of KVSwitch.

There are three major fields in KVSwitch protocol: OP, KEY, and SEQ. OP represents the operation of a query, which can be *Get*, *Set*, *Delete*, or any other possible type used in KVSwitch. KEY stores the key of the item specified by this query. SEQ can be used as a sequence number to ensure reliable transmissions by UDP *Get* queries and other functions (e.g., the version of value). The format of reply packets also follows this specification, save for the different OP fields and payloads.

Packet forwarding is realized using existing routing protocols. The Ethernet and IP headers of a query are set at client according to the data partition to direct the query to the server that owns this item. For hot key queries, KVSwitch may reset the destination address in order to balance the load in a round-robin manner (Section 4.2.2).

At clients and KVS servers, we need to finish the transformation between KVS's API (Application Programming Interface) calls and KVSwitch packets. We develop a client library for clients that facilitates easy access to the KVS following the KVSwitch packet format. The client library translates the API calls made by applications to query packets and also decodes the response packets received from the network.

On each server, we run a simple process that provides two important functions: (i) it maps packets from KVSwitch to API calls for the KVS, and vice versa; (ii) it helps to maintain coherence among replicas in three different situations. The first is sending the hot item on this server to other servers when it is replicated by KVSwitch, the second is updating the value of replicas and sending back a confirm packet when an update request is received from KVSwitch, and the third is deleting a replica when KVSwitch finds that this item is no longer a hot item.

3.2. Processing Query

The processing procedure for incoming packets varies according to their operation types and to whether they are requesting a hot item (see Algorithm 1). Each KVS query first enters the *Hot Key Lookup* module to determine whether the requested item is hot. KVSwitch exerts no influence over queries for cold items other than updating the counting information in the *HashPipe* and *Server Load Count* modules. Alternatively, if this is a query for a hot item, the process varies between the *Get* and *Set* queries, also shown in Figure 3.

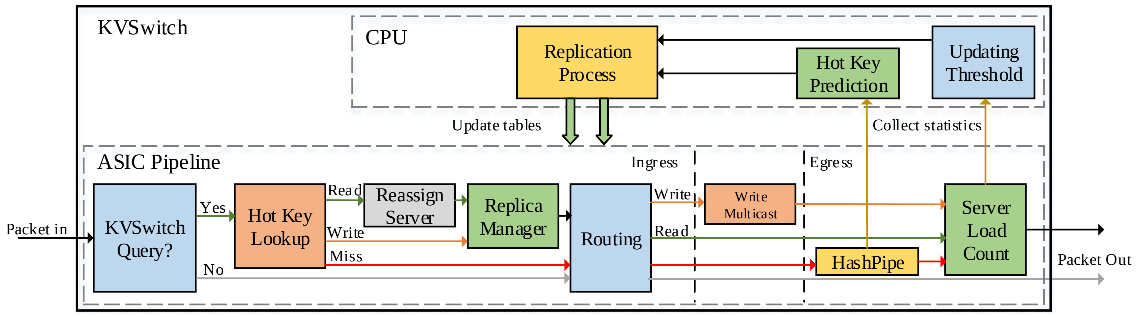


Figure 3. Logical view of KVSwitch.

Algorithm 1: ProcessQuery(pkt)

```
Input: A packet ($pkt$)
1  if $isHotKey(pkt.key)$ then
2      if $pkt.op == Get$ then
3          $s \leftarrow reassignServer(pkt.key)$;
4          if $isValid(s, pkt.key)$ then
5              $pkt.dst \leftarrow serverAddr(s)$;
6          end
7      else if $pkt.op == Set$ then
8          $invalidServer(pkt.key)$ ;
9          $multiCast(pkt, key)$ ;
10 end
11 else
12     $simpleForwarding(pkt)$ ;
13 end
```

Handling *Get* **queries of hot items.** First, the *Hot Key Lookup* and *Reassign Server* modules will assign a new destination server for this packet in a round-robin manner across all servers containing this item (Section 4.2.2). The *Replica Manager* module then checks whether the replica on the new destination server is valid (Section 4.2.3). If valid, the destination IP will be rewritten with the address of the new destination server. Finally, the *Server Load Count* module increases the load on the destination server (Section 4.2.6).

Handling *Set* **queries of hot items.** The core job of handling a *Set* query in KVSwitch is to update all replicas of this key. To do this, the valid flag of this item is first set to invalid on all servers that contain its replicas (Section 4.2.3). The *Set* query itself will be forwarded to its destination server to update the key-value pair as for a normal packet. Secondly, a packet is sent to each server containing a replica of this item to update the value (Section 4.2.4). After the server updates its replica, it sends a confirm packet back to KVSwitch to validate the replica on the corresponding server (Section 4.2.3). Finally, the *Server Load Count* module increases the load on the final destination server in the same way as that for the *Get* queries (Section 4.2.6).

Handling *Delete* **queries of hot items.** *Delete* queries can be treated as special *Set* queries. All valid flags for the item in question are set to false and all servers containing this item will receive a *Delete* packet from KVSwitch to remove the data from KVS. Due to these similarities, we will henceforth focus on discussing *Set* queries and leave out *Delete* ones.

4. KVSwitch Design

In this section, the design and implementation details of KVSwitch are presented.

4.1. Functions in CPU

To cope with dynamic workloads, the set of hot items should be updated frequently according to changes in popularity. In traditional caching systems, a query for an item that is not in the cache would bring that item into cache and evict another item if the cache is full. However, the updating speed of match-action tables is limited and the update rate is insufficient to support traditional mechanisms like LRU (Least Recently Used) and LFU (Least Frequently Used). Instead, KVSwitch uses a different periodic updating approach to set the hot items.

KVSwitch periodically collects statistics from the ASIC pipeline to predict which items are likely to be most popular in the next period (Section 4.1.1). Moreover, server loads in this period are used to dynamically adjust the threshold T (Section 4.1.3). Finally, the decision as to how these hot items should be replicated is made according to their predicted popularity and T (Section 4.1.2).

4.1.1. Load Estimation for Next Period

At the end of each period, the *HashPipe* module records the keys of the K cold items that have been queried most in this segment. The number of queries for the K hot items is also stored in the counter of HotkeyTable (Section 4.2.2). KVSwitch uses these two pieces of information to predict the most popular K items in the next period.

The basis of our predictive approach is the *Time Locality Principle* [2], which points out that access to an item is likely to be continuous in time. Thus, in KVSwitch, we use information from the previous two intervals to predict popularity in the upcoming segment.

Suppose the query count of key x in last period is $L_x^{''}$ and that in this current segment is $L_x^{'}$. Therefore, the load of key x in the next period L_x can be predicted as follows:

$$L_x = \alpha \cdot L_x^{''} + (1 - \alpha) \cdot L_x^{'}, \tag{1}$$

where $\alpha \in [0, 1)$ represents the weight of history information. A higher α means a bigger influence of previous counting statistics. If a key has only been queried in one of the last two segments, the load of the other period is treated as zero.

KVSwitch uses Equation (1) to predict the load for the keys in *HashPipe* and HotkeyTable in this period or $2 \times K$ keys in total. The K keys with higher predicted loads are then selected as the hot keys in the next period.

4.1.2. Item Replication

Once the hot keys have been selected, KVSwitch must determine how to replicate these items and to which storage servers. In KVSwitch, this is a two-step process: the first step is determining how many replicas for each hot item, and the second is deciding where to place these replicas. Optimally solving this can be reduced to a bin-packing problem [18], which becomes intractable for large numbers of keys and servers.

To avoid this complexity, KVSwitch uses a dynamic adapting method to carry out this process. We use a replication threshold T to determine how many times a hot key should be replicated (Section 4.1.3). A predicted hot key with estimated load less than T is considered not hot enough and thus not worthy of being replicated. Conversely, a hot key x with estimated load $L_x > T$ will be replicated $r_x = ceil(L_x/T)$ times. If r_x exceeds the server number M in the rack, r_x is reset to M. Through the filter of T, KVSwitch is able to reduce the number of hot items that need to be replicated.

Except for the original server responsible this hot item, KVSwitch should select $r_x - 1$ servers to store the replica. Here, we number the storage servers form 1 to M and organize them in a ring, as showed in Figure 4 with $M = 8$. We then choose $r_x - 1$ servers to store the replica in the ring from the original server with a step, calculated as $step_x = M/floor(r_x)$.

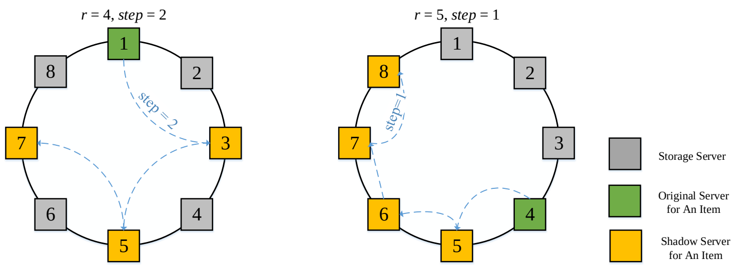


Figure 4. Example of selecting servers for replicas.

In the left figure in Figure 4, a key for which the original server index is 1 is supposed to have $r = 4$ replicasl; thus, $step = 2$. Three $(r - 1)$ servers are chosen according to the $step$, which are servers 3, 5, and 7. The situation where $r = 5$ is also shown in the right figure in Figure 4, and a key from server 4 is replicated with $step = 1$. Therefore servers 5–8 are selected consecutively to store the replica.

After the hot keys and their replication strategies have been selected, the CPU updates the match-action tables in the ASIC pipeline and resets the register arrays. Packets are then sent to the servers originally responsible for the hot keys that push the hot items to the servers assigned to hold the replicas.

4.1.3. Adaptive Threshold

The threshold T determines both the number of keys that will be replicated and their replication times. Thus, it has a dominant effect on how even the balance achieved by KVSwitch will be. It is obvious that a smaller value of T results in more replicas and, in theory, a more balanced workload. However, as the number of replicas increases, each server requires more resources to maintain the replicas and more overhead to maintain consistency. Accordingly, rather than using a fixed T, KVSwitch automatically calculates T based on the real-time imbalance across servers, as described below.

At the beginning, T has a preset value T_0, which is relatively large. Considering the replication process for interval $i + 1$, first, the load statistics on servers collected from the ASIC pipeline are analyzed. If the servers exceeds an administrator-set bound (e.g., the server with the highest load should see at most 30% more queries than average), then KVSwitch reduces T to obtain a better balance. The threshold in interval $i + 1$ T_{i+1} is calculated using an attenuation coefficient $\gamma : T_{t+1} = \gamma * T_i$. If not, then T stays unchanged and is used for replication.

If the statistics for period $i + 1$ still show imbalance, KVSwitch reduces T again. The reduction stops once a promising balance is achieved. Using this dynamic adjusting mechanism, KVSwitch can set an appropriate T according to the workload to achieve balance with minimal replicas.

4.2. Functions in ASIC pipeline

4.2.1. Programming with ASIC Pipeline

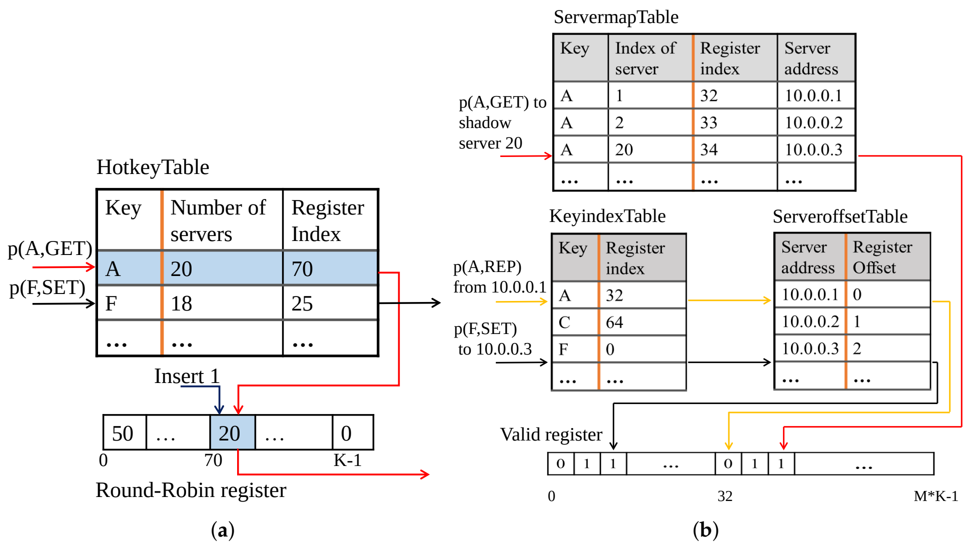
The ASIC pipeline in the programmable switch consists of three main components: ingress pipelines, a traffic manager, and egress pipelines. A switch usually has multiple ingress and egress pipelines but only one traffic manager. An incoming packet first enters a programmable parser, which parses the packet header. The header is then processed in the ingress pipeline. The packet header is queued in the traffic manager after coming out from the ingress. After being scheduled by the traffic manager, it is sent into the egress pipeline and emitted via an egress port.

Both the ingress and egress pipelines have a list of multiple stages. These stages process packets in a sequential order. Each stage has its exclusive resources, including match-action tables and register arrays. The match-action tables are also executed sequentially, which means tables with dependency cannot be placed in the same stage, while the tables in the same stage can be executed concurrently. Registers can be used to save the information and pass it between different packets, but the registers in one stage are only accessible by this stage. Packet headers, along with metadata, flow through the stages; thus, they are often designed to store interim information for a specific packet and to pass it to subsequent stages.

Developers use a Domain Specific Language (DSL) like P4 [19] to specify the packet processing in the ASIC pipeline. We are able to build a custom parser, to design match-action tables, to create registers, and to define the metadata format using P4. The main challenge is expressing a function with a match-action model under the limitation of the ASIC pipeline's hardware constraints. Firstly, the number of stages in the pipeline is limited; thus, if a program contains too many tables and dependencies, it may not be able to be placed into the pipeline. Secondly, in each stage, the resources are limited (primarily the memory). Tables and registers cannot claim memory that exceeds this limit.

4.2.2. Round-Robin Load Balancer

KVSwitch uses one match-action table and a register array to implement the load balance process, as shown in Figure 5a.



HotkeyTable

Key	Number of servers	Register Index
A	20	70
F	18	25
...	...	...

ServermapTable

Key	Index of server	Register index	Server address
A	1	32	10.0.0.1
A	2	33	10.0.0.2
A	20	34	10.0.0.3
...	...	...	...

KeyindexTable

Key	Register index
A	32
C	64
F	0
...	...

ServeroffsetTable

Server address	Register Offset
10.0.0.1	0
10.0.0.2	1
10.0.0.3	2
...	...

Figure 5. ASIC designs of two important modules. (**a**) Load balance module; (**b**) Consistency module.

All packets identified as KVS queries first look up the HotkeyTable to check whether the query is requesting a hot item. HotkeyTable matches the *KEY* field in the packet header and returns two parameters: the number of servers containing this item and an index of a cell in the *Round-Robin* register array. Both parameters are stored in self-defined metadata, which is in a one-to-one relationship with the packets. The *Round-Robin* register array contains *K* cells, each of which is used for a hot key in HotkeyTable.

If this packet is a *Get* query, KVSwitch will assign a new destination server to balance the rack. KVSwitch uses the index obtained from HotkeyTable to obtain the corresponding value in the *Round-Robin* register array and also stores it in the metadata, which denotes the new destination server among all shadow servers of this key. The value in this cell is then incremented by 1 so that the next query for this key is redirected to the next server in a *Round-Robin* manner. When this value exceeds the total number of servers containing this item, it is reset to 1.

Otherwise, if this is a *Set* query, no further operation with the *Round-Robin* registers is needed, so it goes directly to the following tables (Section 4.2.3).

Some examples are shown in Figure 5a. A *Get* query for hot key A hits the HotkeyTable, and the two parameters are 20 and 70. This means that, in the rack, there are 20 servers containing the required item and that we should look up the 70th cell of the *Round-Robin* register array to find the new destination server. Since the value in the 70th cell is 20, this query will be redirected to the 20th server containing this item. Because 20 equates to the total number of servers that contain item A, the new value in this cell will be set to 1 to redirect the next *Get* query for A to the first server. A *Set* query is also tested with HotkeyTable to check whether it will modify a hot item (see *Set* query for F in Figure 5a). The action parameters are written into the metadata in the same way as for *Get* queries, but no further operation is executed in this process.

4.2.3. Valid Check and Replica Consistency

The *Replica Manager* module is in charge of checking the validation and of maintaining the consistency of replicas across storage servers. For *Get* hot key queries, it will check whether the replica on the new server assigned in the *Round-Robin* register array is valid. For *Set* queries, *Replica Manager* should update the valid replica flags for this item.

The *Replica Manager* module consists of three tables and a *Valid* register array, as shown in Figure 5b. The *Valid* register array stores the valid flag for all replicas, such that it has $M \times K$ cells in total. The *Valid* register array is segmented into K parts; from the cell with an index of zero, every M cell is assigned to represent the replicas of a key on the M servers. ServermapTable is used for *Get* queries to obtain the real IP address of the newly assigned server. Both KeyindexTable and ServeroffsetTable help to find the location of the valid flag of a hot item on a specified server in the *Valid* register. KeyindexTable gives the index of the first register cell belonging to a certain key's replica. The ServeroffsetTable table specifies the offset of the cell belonging to a certain server in the *Valid* register array.

Three examples to illustrate the working process of *Replica Manager* are presented in Figure 5b.

(1) A *Get* query for key A sent to the 20th server containing the corresponding item: ServerMapTable matches the key and index of the server, after which the action parameters (34 and 10.0.0.3) are stored in metadata. The register index 34 denotes the location of the valid bit of the replica on the new destination server in the *Valid* register array. Since the valid bit in the 34th cell in the Valid register is 1, the replica of A on server 10.0.0.3 is deemed to be valid. Finally, the new IP address 10.0.0.3 is written into the packet header; thus, this query is redirected. Otherwise, if the valid bit is 0, this packet would be sent to its original destination server without being redirected.

(2) An update response for key A from server 10.0.0.1: When a server receives an update request for an item (unless it is the original server of the item), it will inform KVSwitch with a reply packet after the update process is finished (Section 4.2.4). This packet from 10.0.0.1 first matches the KeyindexTable and obtains the beginning index in the *Valid* register of this item (here, 32). ServeroffsetTable then matches the exact server address and returns the stride for this special server (in this case, 0). The final position of the valid bit for this particular item and server is calculated by adding the two results together (in this case, the result is 32). Accordingly, the valid bit in the 32nd cell is set to 1, which indicates that the replica of A on server 10.0.0.1 is now valid.

(3) A *Set* query for key F, sent to its original server 10.0.0.3: In this situation, KeyindexTable is used to obtain the beginning index in the *Valid* register of this key. Here, the valid register cells for key F start at 0. We then need to locate the register cell for the original server, which is given by ServeroffsetTable (here, this is 2 for 10.0.0.3). Since the *Set* query is sent via TCP packet, it is reliable in most cases. Thus, KVSwitch sets all replicas to invalid except that on the original server. For this packet, KVSwitch sets all cells with indices from 0 to $M - 1$ to 0 except for the 2nd cell.

The processing in ingress pipeline ends here.

4.2.4. Multicast *Set* Query

Multicast is a basic function in switches realized in the traffic manager between the ingress and egress pipelines. It can generate many instances of a packet and sends them to the egress pipeline. *Multicast* retains a large number of groups; in multicasting, a packet should first choose a group in the ingress pipeline based on some criteria, after which the traffic manager replicates the packet, sets the egress port for each according to preset information for the group, and sends them to the egress pipeline [20].

In KVSwitch, we have two tables cooperating with the traffic manager to inform servers that they should update their replicas when processing a *Set* query of a hot item. These tables are MultigroupTable and PorttoserverTable. Each hot key marks a multicast group, and the replicated packets are sent to the preset egress ports. The group identification is conducted using MultigroupTable, which matches the *KEY* in the packet header and returns the group ID. In the traffic manager, this packet is replicated and set with egress ports according to its group. Finally, PorttoserverTable in the egress pipeline matches the egress port and rewrites the destination IP address in the header based on the match result. It should be noted that since the original *Set* query is sent as a TCP packet, each subsequent instance of a multicast packet is also in TCP. Although the servers (except for the original server) do not maintain a TCP connection with the client, servers can decode the packet based on the KVSwitch protocol via the transform layer.

4.2.5. Time-Segmented Top-K Key Tracker

KVSwitch uses the HashPipe data structure [21] to efficiently track the Top-K most popular cold keys in each time interval. HashPipe takes advantage of the high throughput of the ASIC pipeline to count the packets without sampling. Here, hot keys are not considered, since the HotkeyTable has a counter for each entry and thus every hot key query is counted accurately.

HashPipe consists of a list of d hash tables such that each table j has a unique hash function h_j. Moreover, each table has $w = K/d$ slots; thus, a Hashpipe can track a total of the K hottest keys. The tables in HashPipe are implemented using registers, meaning that it is possible to inspect every query for a cold key since the control plane is not involved when the tables are changed. A hash table has two register arrays, each of which has w elements, used to store the key and associated count respectively.

The basic idea behind HashPipe is to insert the newly arrived key into tables and to remove the key with the smallest counter. In this way, HashPipe is able to retain the K cold keys with bigger counters. To accomplish this, each query for cold items contains the lowest counter ever seen and the corresponding key in metadata, which flows through the tables in HashPipe.

For a KVS query of a cold item, HashPipe first treats it as a new key that has never been seen before. The key of this packet, along with a counter-value of 1, is written into metadata. At each hash table j, the key carried with the current packet is hashed by h_j to the slot l_j. If the key matches in slot l_j (i.e., the l_j element in register array), then the count in l_j is updated by adding the count being carried in the metadata. Alternatively, if the slot is empty, the carried key and count are stored in this slot. Both situations above end the processing of HashPipe. Otherwise, the key and count corresponding to the larger of the counters that is carried and the one in the slot are written back into the table, and the smaller one is carried on the packet to the next hash table. Through this process, keys with larger counters are saved, while those with smaller counters are weeded out.

Consider a newly arrived packet with a counter of one. It is possible that, in each hash table, this packet will find a larger count in the slot. As a consequence, this key will not be retained in HashPipe. The next packet of this key will then undergo the same process, which means this key will be neglected. To prevent this, HashPipe chooses to always insert the key in the first table and to evict the existing key and counter to metadata, thus ensuring that this key has been taken into consideration.

An example that illustrates how HashPipe works is presented in Figure 6 [21]. A packet with a key K and counter of 1 enters the HashPipe and is hashed to slot 2 in the first table. Since this is the first table, K is inserted and B is carried in metadata. In the second hash table, key B is hashed to the slot containing key E; however, since the count of B is larger than that of E, B is written into the slot and E is carried out in metadata. Finally, since the count of L (which E hashes to) is larger than that of E, L stays in the table and E is dropped (as this is the last table). The net effect is that the new key K is inserted in HashPipe and the minimum of the three keys B, E, and L is evicted.

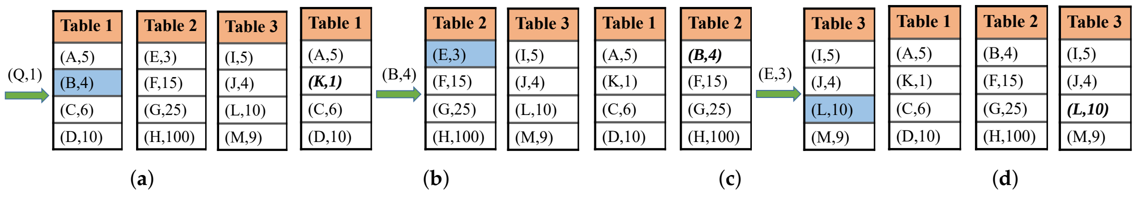


Figure 6. An illustration of HashPipe in KVSwitch. (**a**) Initial state; (**b**) New key is placed with value 1 in first stage; (**c**) B being larger the E; (**d**) L is retained in HashPipe.

4.2.6. Server Load Statistics

For each KVS packet, regardless of whether it is for a hot key, KVSwitch records its destination server before it goes out the egress port in order to keep track of the server load. A LoadcountTable of M entries is used to count the total number of packets that have been sent to M servers in this period. Every time a packet forwarded to server j is detected, LoadcountTable matches the destination IP and the counter of the matched entry is incremented by one to record this query.

5. Evaluation

5.1. Experiment Setups

Testbed. The testbed consists of one 3.2 Tbps Wedge 100BF-32X switch with a Tofino chip and three Dell PowerEdge R820 servers. Each server has two Xeon E5-4603 CPUs, 256 GB memory, and an Intel XL710-QDA2 network interface card (NIC) with two 40 G ports. The CPU in the Tofino switch is an Intel Pentium D-1517 CPU with four cores running at 1.6 GHz. We assign two servers to be KVS servers and another one to be the client for generating queries; all of them are connected to the switch via two 40 G cables. Limited by the servers we have, we emulate M storage servers by using $M/2$ queue on each server machine. Each queue process queries for one KVS partition as a server and drops queries if the received queries exceed its processing rate.

Implementation. The KVSs on the servers are realized by running memcached services. Both the client library and the server agent are implemented in Lua with Moongen [22] for optimized throughput performance. Inside the Switch, functions in the ASIC pipeline are written in P4 and the other parts are written on CPU in Python. The ASIC pipeline communicates with the CPU through Thrift APIs, which are generated by the switch compiler.

Traces. We generated the query traces following the Zipf distribution as skewed workloads, and the items in KVS are partitioned across storage servers. Three different skewness parameters are used for generating, 0.9, 0.95, and 0.99, which are commonly used in the literature [8,9,12]. In our experiments, the client keeps sending queries for 5 min at a maximum sending rate of about 30 MQPS (Million Queries Per Second) for every test. Most experiments use read-only workloads, and write queries are tested particularly. By default, K and M are set to 10,000 and 32, respectively. The sizes of the key and value are 16 and 128 bytes, respectively.

Methodology. The main evaluation metric we use is the imbalance factor of the rack λ defined in Equation (2), which is also used in the literature [12,23]:

$$\lambda = \sum_{j=1}^{M} \left(\frac{|L_j - \bar{L}|}{\bar{L} * M} \right), \tag{2}$$

where L_j is the KVS load on server j and $\bar{L}$ is the average number of queries across all servers. The value of λ ranges from $[0, 2 - \frac{2}{M}]$, and a larger λ signifies a more imbalanced load distribution.

The baseline we compare to is the KVS system without any balancing techniques, marked as *Original*. Moreover, we also compare the balancing performance between KVSwitch and NetCache [8], with which the hot items are cached in ASIC pipeline in programmable switches to balance the KVS.

5.2. Performance of Load Balance

Performance of load balance. Figure 7a shows the system imbalance factor under different skewness conditions with read-only queries. Here, we extend the skewness to 1.5 to check KVSwitch under extremely skewed workloads. As expected, the imbalance factor rises quickly with no balancing techniques. However, in contradistinction to the rise, both KVSwitch and NetCache effectively control the imbalance factor around 0.017 even when the workload skewness is 1.5. More specifically, for traces with skewnesses of 0.9, 0.95, and 0.99, the imbalance factors with KVSwitch are 0.015, 0.013, and 0.017 respectively. These results show that KVSwitch is able to keep the storage servers balanced effectively.

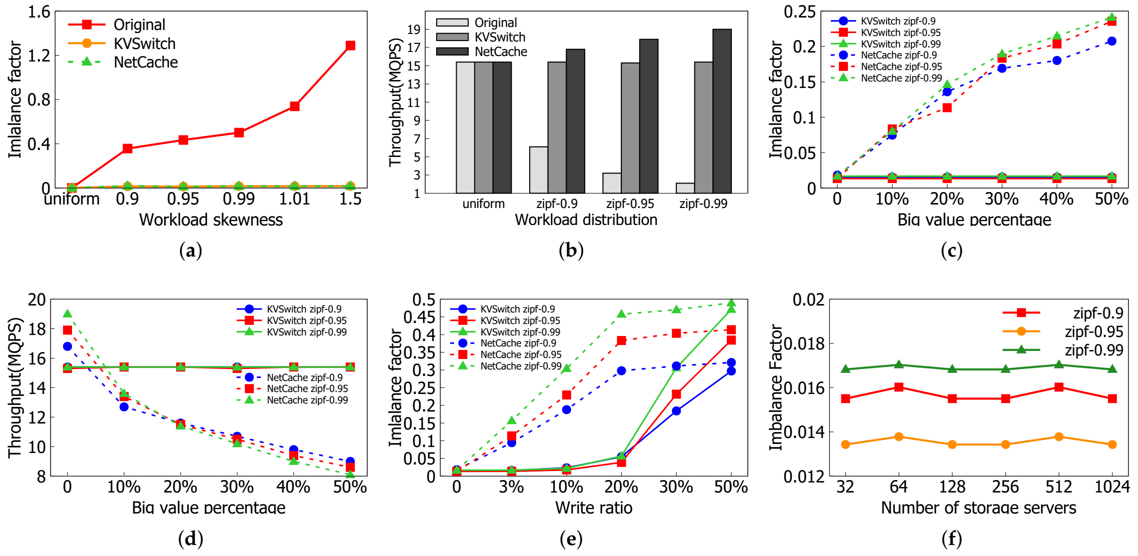


Figure 7. KVSwitch performance of load balance. (**a**) Effect of balancing techniques; (**b**) Throughput vs. skewness; (**c**) Imbalance factor vs. value size; (**d**) Throughput vs. value size; (**e**) Impact of write ratio; (**f**) Scalability(simulation).

For comparison from another angle, Figure 7b shows the system throughput with different techniques. When the workload is under uniform distribution, three systems show no differences and all achieve the maximum throughput of the storage servers (15.4 MQPS). As the workload gets more skewed, throughput of original KVS decreases quickly from 15.4 to 2.1 MQPS with a skewness of 0.99. In contrast, since the workload is balanced by KVSwitch to all servers, the throughput does not decline. Once deployed in ASIC pipeline, KVSwitch works at a line speed (3.2 Tbps for our switch) which matches the throughput of hundreds of SLBs. As for NetCache, since the queries of hot items are all handled by the switch and the switch throughput is higher than the storage server, the system throughput increases as the workload gets more skewed.

Impact of value size. Due to the limited memory in ASIC pipeline, NetCache does not cache big items of which the values exceed 128 bytes in the pipeline. Considering KVS providers report an increasing ratio of these big values in real KVS [24], the balancing techniques should also be effective in this situation.

In Figure 7c,d, we compare NetCache and KVSwitch in the situation when the workload contains big values. It is noted that the λ increases significantly with NetCache when the percentage of big values grows (under the skewness of 0.99, from 0.016 to 0.24). At the same time, the throughput drops from 19.0 to 8.1 MQPS rapidly. However, KVSwitch does not manipulate the value field of items; thus, the balancing effect is not affected by big values.

Impact of write ratio. As the read queries of hot keys are dispersed across servers by balancing techniques, the write queries can offset the benefits. Figure 7e plots how λ of the two systems changes under different write ratios. The write queries are chosen randomly from all queries based on the write ratio.

While both techniques are suffered, λ increases much slower with a write ratio under 20% with KVSwitch. After this, the balancing effect drops significantly; both techniques show almost the same λ with the original KVS system when the write ratio achieves 50%. Since the read-intensive workloads are common in many real-world scenarios (the write ratio is reported as 0.03 for Facebook [1]), KVSwitch shows greater potential than NetCache for load balance in real life.

Scalability. This experiment evaluates how KVSwitch can potentially scale to multiple racks. We test whether KVSwitch can balance across servers in different racks (the coherence of replicas is left for future work). Figure 7f shows the λ when scaling the KVS system to 1024 servers on 32 racks. Here, the environment is simulated by creating multiple switches with a Fat-Tree topology and the leaf switch is the ToR switch. Each ToR switch connects to one Mininet host acting as a rack containing 32 servers. In a KVS system with multiple racks, the workload is balanced first across racks and then across the servers inside each rack. Thus, the spine switches distribute the queries across leaf switches, while the latter balances the load across servers. Simulation results show that KVSwitch maintains workload balance even when scaled to multiple racks, as λ remains under 0.018.

5.3. Accuracy of Hot Key Prediction

In this part, we focus on testing whether KVSwitch's prediction module can successfully identify the most popular K keys in the next period. The metric used here is the *overlap ratio* of the predicted and real most popular K keys. The prediction accuracies of all intervals are collected, after which the average ratio is calculated to determine the final accuracy.

KVSwitch uses two types of information to carry out its predictions: accurate counting for keys in HotkeyTable and inaccurate counting from HashPipe. Thus, records of $2 \times K$ keys are gathered to identify the hottest K keys.

As Figure 8a demonstrates, KVSwitch predicts more accurately when K is smaller. When K increases from 1000 to 10,000, the accuracy drops from 95.1% to 51.7% under a skewness of 0.99. This result comes from the characteristic of the Zipf distribution in which the frequency of a key is inversely proportional to its rank among all keys. Thus, identifying a hot key with a lower rank from all keys is far more difficult, as the difference of counts between this key and other keys is smaller. In a similar way, prediction is easier for a workload with a higher skewness. However, this diversity is no longer observed with a larger K, as the influence of varying skewness is mainly seen on the keys with higher ranks.

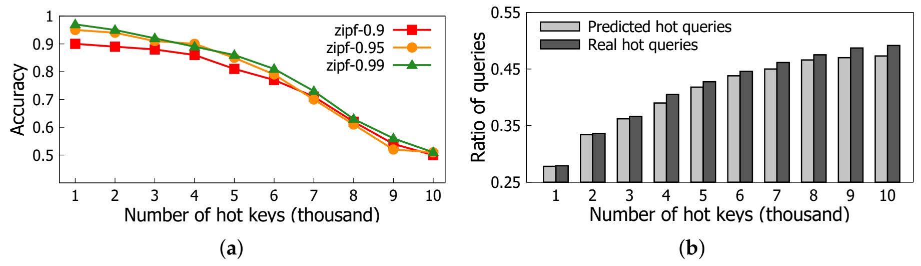


Figure 8. Accuracy of predicting popular keys. (**a**) Accuracy of finding K most popular keys; (**b**) Percentage of workload of hot keys.

Another straightforward comparison is shown in Figure 8b. Here, we study the ratio of queries for the predicted hot keys of all queries. The workload here follows a Zipf-0.99 distribution. The queries of the predicted K hot keys are in almost the same proportions as the real K hottest keys (with K = 1000, 27.8% and 27.9%, respectively). Even though the accuracy of prediction falls as K becomes larger, the ratio of queries is still very close (with K = 10,000, 47.3% and 49.1%, respectively). This is because, even if the KVSwitch predicts an incorrect hot key with a low rank, it has little difference in query frequency with the right hot key.

5.4. Dynamic Change of the Threshold

In this section, we study the effect of the dynamic threshold mechanism in KVSwitch. We first compare λ under KVSwitch and the fixed replica numbers and then detail the changes in the number of hot keys and replicas on each server caused by T.

Figure 9a presents a comparison between two strategies: replicating hot keys to all servers and using KVSwitch. Compared to replicating hot keys to all servers, which is used in production [3], KVSwitch can obtain a comparable λ (0.168 and 0.164 for skewness = 0.99). This suggests that using KVSwitch does not result in any balancing degrade compared with traditional replication-based techniques. The benefit of T can be seen in two aspects: the number of hot keys the switch needs to hold (even if it is set to K) and the number of replicas on storage servers. Smaller values of these two numbers indicate reduced memory consumption in the switch and servers, respectively.

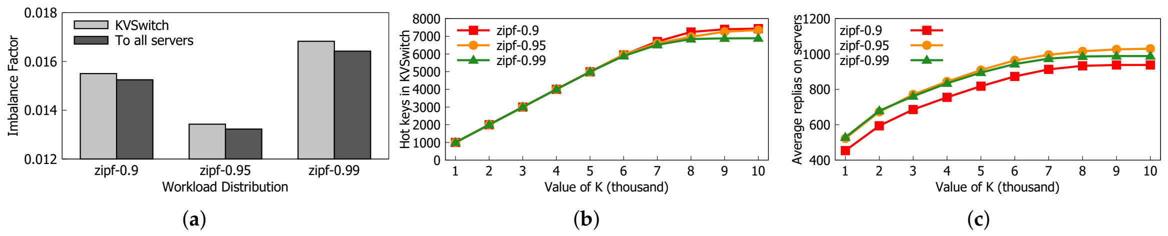


Figure 9. Dynamic change of the threshold. (**a**) The change of the threshold; (**b**) Number of hot keys. (**c**) Number of replicas.

Moreover, Figure 9b shows that KVSwitch will not replicate a predicted hot key if there is little resultant benefit on load balance. For a preset K less than 6000, KVSwitch simply keeps K hot keys in its table; however, when K becomes as large as 10,000, only about 7000 keys are judged worthy to be replicated by T. The unnecessary 3000 keys are filtered, thus reducing memory usage in the ASIC pipeline.

The storage servers also benefit from T due to fewer replicas being kept. In the situation where a hot item is replicated to all servers, each server is required to keep K replicas, which in turn increases the overhead involved in updating replicas. Figure 9c shows that KVSwitch can significantly reduce the number of replicas on storage servers. As K grows larger, the average number of replicas on servers increases slowly (from 529 to 988 with skewness of 0.99). Compared with keeping all K hot items on each server, KVSwitch significantly reduces memory usage by about 90.12%.

5.5. Feasibility Verification

Due to the limited hardware resource in switches, especially for the ASIC pipeline, we also need to evaluate the resources required by KVSwitch in order to determine its feasibility. In Table 1, we compare the hardware resources required by KVSwitch (K = 10,000, M = 32) to that required by switch.p4 and NetCache (K = 10,000). switch.p4 [25] is a P4 program that implements most networking features (L2/L3 forwarding, VLAN, QoS, etc.) for a typical data-center ToR switch.

Table 1. Hardware resource cost in ASIC pipeline.

Resource	Switch.p4	KVSwitch	KVSwitch + Switch.p4	NetCache
Match Crossbar	50.13%	7.72%	57.85%	45.34%
Hash Bits	32.35%	17.58%	49.93%	30.54%
SRAM	29.79%	38.43%	68.22%	48.98%
TCAM	28.47%	0.00%	28.47%	34.21%
VLIW Actions	34.64%	8.26%	42.9%	37.85%
Stateful ALUs	15.63%	24.61%	40.24%	16.74%

SRAM: Static Random Access Memory; TCAM: Ternary Content Addressable Memory; VLIW: Very Long Instruction Word; ALU: Arithmetic Logic Unit.

We note that KVSwitch consumes relatively larger proportions of hash bits, SRAM and stateful ALUs (17.58%, 38.43% and 24.61%, respectively). SRAM is used for the tables and registers. Hash Bits and Stateful ALUs are mainly consumed by HashPipe and conditional operations for registers. Also, the combined usage of all resources by switch.p4 and KVSwitch remains below 70%. This means that KVSwitch can easily fit on top of switch.p4. Moreover, KVSwitch consumes much less SRAM compared to NetCache (29.79% vs. 48.98%) at the cost of little increases in demands for other resources. In other words, KVSwitch can save more resources for other network functions compared to NetCache.

In a KVS containing 100 million items, the 10,000 items with the highest ranks account for about 49.15% of all queries under a Zipf distribution (skewness = 0.99). Therefore, KVSwitch with K = 10,000 would be able to balance a workload for a large KVS in real life.

We also evaluate the increase in packet forwarding latency introduced by KVSwitch. Delightfully, packet latencies in original KVS and KVSwitch are 185 and 214 nanoseconds by average, respectively. The overhead in latency is only 15.7%, and this result reveals the validation of KVSwitch in the other aspect.

6. Related Work

In-Memory Key Value Stores. Given the high-throughput and low latency requirements of large-scale Internet services, a large number of works have focused on optimizing and improving in-memory KVS [1,26–32]. The techniques used in these works have varied from using new data structures and algorithms to exploiting different system-level optimizations and new hardware capabilities. In a departure from our goal of balancing the workload, these works focus on how to maximize the performance of an in-memory KVS.

Load Balancing. The performance of a large-scale KVS is often bottlenecked by overloaded servers due to highly skewed workloads in real life. Some works rely on consistent hashing [33] and virtual servers [34] to mitigate load imbalance. One major limitation of these solutions is their inability to deal with dynamic workload changes. Data replication [12,16,35] is conceptually straightforward and, thus, often used by KVS providers for load balance. However, challenges in data replication include determining the appropriate level of replication granularity (item, partial shard, or entire shard), maintaining consistency among replicas, and matching the high throughout of modern KVS. Compared to previous replication methods, KVSwitch deals with these challenges more simply and efficiently by using programmable switches.

Caching can be an effective dynamic load balancing technique for distributed KVS. A fronted cache can absorb hot key queries and reduce the skew of the load across the back-end servers. There are various approaches based on this idea, including (i) placing a cache at the front-end load balancer [10], (ii) using a programmable switch to redirect hot key queries to the cache node [9], and (iii) using the memory of a programmable switch as a cache node [8]. The fundamental shortcoming of caching approaches is the capability limit of the cache server, including IO performance, processing ability, and memory capacity. As shown in our experiments, KVSwitch can avoid these limitations and can balance workloads with different characteristics.

Programmable Switches. Due to increased control requirements for switch behavior imposed by major cloud providers [36], switch vendors [13,37,38] have begun to expose low-level hardware primitives of high-speed, low-cost ASICs to customers. Except for the advent in programmable hardware, programming languages [17,39] for data plane have been more expressive. A large amount of work has been proposed to facilitate realizing more functionality in programmable switches, from traditional network monitoring [40,41] to new network applications [42,43]. We expect the switches can be treated as another computing device like graphics processing units (GPUs) inside the datacenter network in the future.

7. Conclusions and Future Work

In this paper, we present KVSwitch, a switch-based load balancer for KVS that requires no additional hardware. KVSwitch aims to fully exploit the potential of emerging programmable switches in order to realize the load balance as a network function. Meanwhile, taking advantage of the fact that the switch controls all KVS queries, KVSwitch is able to adapt its replicating strategies intelligently according to the status of KVS. To overcome hardware limitations, we decompose KVSwitch and fit it into both the ASIC pipeline and the CPU via careful design. We prototype KVSwitch in a Tofino switch. Experimental results show that KVSwitch can effectively keep the KVS balanced even under highly skewed workloads and that the resource usage on servers is also minimized. Even KVSwitch is dedicated to solving the imbalance in distributed KVS; the idea of it can be used to improve the performance of many other applications, like web searching and cloud storage services.

Currently, KVSwitch is only designed for the ToR switch. In the future, we plan to scale it out to multiple racks for large-scale deployments. This requires us to redirect KVS queries in higher-level switches, e.g., spine switches. Another open issue for KVSwitch is improving its' performance under write-intensive workloads. A preliminary idea is performing write operations in the switch firstly and writing it back to storage servers afterwards. To do this, we need to design a mechanism to manage the updated hot items carefully.

Author Contributions: Conceptualization, Y.S., J.F., and M.W.; methodology, Y.S. and J.F.; software, Y.S. and J.F.; validation, Y.S., J.F., and C.Z.; writing—original draft preparation, Y.S.; writing—review and editing, Y.S. and M.W.; supervision, M.W. and C.Z.

Funding: This research was funded by theNational Key Research and Development program under No.2016YFB1000400; the National Key Program JZX2017-1585/Y479; and the National Nature Science Foundation of China under NSFC No. 61872377, 61802417, 61802420, and 61502509.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

KVS	Key-value Stores
SLB	Software Load Balancer
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
ToR	Top-of-Rack
RMT	Reconfigurable Match-action Tables
API	Application Programming Interface
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
LRU	Least Recently Used
LFU	Least Frequently Used
DSL	Domain-Specific Language
P4	Programming Protocol-Independent Packet Processors
NIC	Network Interface Card

QPS	Queries per Second
ALU	Arithmetic Logic Unit
SRAM	Static Random Access Memory
TCAM	Ternary Content-Addressable Memory
VLIW	Very Long Instruction Word
QoS	Quality of Service
GPU	Graphics Processing Unit

References

1. Nishtala, R.; Fugal, H.; Grimm, S.; Kwiatkowski, M.; Lee, H.; Li, H.C.; McElroy, R.; Paleczny, M.; Peek, D.; Saab, P.; et al. Scaling Memcache at Facebook. In Proceedings of the NSDI, Lombard, IL, USA, 2–5 April 2013; Volume 13, pp. 385–398.
2. Denning, P.J. The Locality Principle. In *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*; World Scientific: Singapore, 2006; pp. 43–67. [CrossRef]
3. Alibaba. Tair Key-Value System in Alibaba. 2018. Available online: https://m.aliyun.com/yunqi/articles/316466? (accessed on 7 November 2018).
4. Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, El Paso, TX, USA, 4–6 May 1997; ACM: New York, NY, USA, 1997; pp. 654–663. [CrossRef]
5. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; ACM: New York, NY, USA, 2010; pp. 143–154. [CrossRef]
6. Huang, Q.; Gudmundsdottir, H.; Vigfusson, Y.; Freedman, D.A.; Birman, K.; van Renesse, R. Characterizing load imbalance in real-world networked caches. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks, Angeles, CA, USA, 27–28 October 2014; ACM: New York, NY, USA, 2014; p. 8. [CrossRef]
7. Novakovic, S.; Daglis, A.; Bugnion, E.; Falsafi, B.; Grot, B. An Analysis of Load Imbalance in Scale-out Data Serving. In *ACM SIGMETRICS Performance Evaluation Review*; ACM: New York, NY, USA, 2016; Volume 44, pp. 367–368. [CrossRef]
8. Jin, X.; Li, X.; Zhang, H.; Soulé, R.; Lee, J.; Foster, N.; Kim, C.; Stoica, I. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017; ACM: New York, NY, USA, 2017; pp. 121–136. [CrossRef]
9. Li, X.; Sethi, R.; Kaminsky, M.; Andersen, D.G.; Freedman, M.J. Be Fast, Cheap and in Control with SwitchKV. In Proceedings of the NSDI, Santa Clara, CA, USA, 16–18 March 2016; pp. 31–44.
10. Fan, B.; Lim, H.; Andersen, D.G.; Kaminsky, M. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In Proceedings of the 2nd ACM Symposium on Cloud Computing, Cascais, Portugal, 26–28 October 2011; ACM: New York, NY, USA, 2011; p. 23. [CrossRef]
11. Gavrielatos, V.; Katsarakis, A.; Joshi, A.; Oswald, N.; Grot, B.; Nagarajan, V. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; ACM: New York, NY, USA, 2018; p. 21. [CrossRef]
12. Zhang, W.; Wood, T.; Hwang, J. Netkv: Scalable, self-managing, load balancing as a network function. In Proceedings of the 2016 IEEE International Conference on Autonomic Computing (ICAC), Wurzburg, Germany, 17–22 July 2016; pp. 5–14. [CrossRef]
13. Barefoot. Tofino: Programmable Switch up to 6.5 Tbps. 2017. Available online: https://barefootnetworks.com/products/brief-tofino/ (accessed on 7 November 2018).
14. Yang, S.; Jiawei, F.; Mei, W.; Chunyuan, Z. KVSwitch: An In-network Load Balancer for Key-Value Stores. In Proceedings of the 2019 IEEE Symposium on Computers and Communications (ISCC), Barcelona, Spain, 30 June–3 July 2019.
15. Yang, Y.; Zhu, J. *Write Skew and Zipf Distribution: Evidence and Implications*; ACM: New York, NY, USA, 2016; Volume 12, p. 21. [CrossRef]

16. Hong, Y.J.; Thottethodi, M. Understanding and mitigating the impact of load imbalance in the memory caching tier. In Proceedings of the 4th annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; ACM: New York, NY, USA, 2013; p. 13. [CrossRef]
17. Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al. *P4: Programming Protocol-Independent Packet Processors*; ACM: New York, NY, USA, 2014; Volume 44, pp. 87–95. [CrossRef]
18. Berenbrink, P.; Friedetzky, T.; Hu, Z.; Martin, R. *On Weighted Balls-Into-Bins Games*; Elsevier: Amsterdam, The Netherlands, 2008; Volume 409, pp. 511–520. [CrossRef]
19. Group, P. Behavioral Model. 2017. Available online: https://github.com/p4lang/behavioral-model (accessed on 7 November 2018).
20. Group, P. Multicast in P4 Behavioral Model. 2018. Available online: https://github.com/p4lang/tutorials/issues/22 (accessed on 7 November 2018).
21. Sivaraman, V.; Narayana, S.; Rottenstreich, O.; Muthukrishnan, S.; Rexford, J. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*; ACM: New York, NY, USA, 2017; pp. 164–176. [CrossRef]
22. Emmerich, P.; Gallenmüller, S.; Raumer, D.; Wohlfart, F.; Carle, G. MoonGen: A Scriptable High-Speed Packet Generator. In Proceedings of the Internet Measurement Conference 2015 (IMC'15), Tokyo, Japan, 28–30 October 2015. [CrossRef]
23. Pearce, O.; Gamblin, T.; De Supinski, B.R.; Schulz, M.; Amato, N.M. Quantifying the effectiveness of load balance algorithms. In Proceedings of the 26th ACM International Conference on Supercomputing, Venice, Italy, 25–29 June 2012; ACM: New York, NY, USA, 2012; pp. 185–194. [CrossRef]
24. Lai, C.; Jiang, S.; Yang, L.; Lin, S.; Sun, G.; Hou, Z.; Cui, C.; Cong, J. Atlas: Baidu's key-value storage system for cloud data. In Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 30 May–5 June 2015; pp. 1–14. [CrossRef]
25. Group, P. A P4 Data Plane of an L2/L3 switch. 2018. Available online: https://github.com/p4lang/switch/tree/master/p4src (accessed on 7 November 2018).
26. Dragojević, A.; Narayanan, D.; Hodson, O.; Castro, M. FaRM: Fast remote memory. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, Seattle, WA, USA, 2–4 April 2014; pp. 401–414.
27. Fan, B.; Andersen, D.G.; Kaminsky, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In Proceedings of the NSDI, Lombard, IL, USA, 2–5 April 2013; Volume 13, pp. 371–384.
28. Kalia, A.; Kaminsky, M.; Andersen, D.G. Using RDMA Efficiently for Key-Value Services. In *ACM SIGCOMM Computer Communication Review*; ACM: New York, NY, USA, 2014; Volume 44, pp. 295–306. [CrossRef]
29. Kaminsky, A.K.M.; Andersen, D.G. Design guidelines for high performance RDMA systems. In Proceedings of the 2016 USENIX Annual Technical Conference, Denver, CO, USA, 22–24 June 2016; p. 437.
30. Li, S.; Lim, H.; Lee, V.W.; Ahn, J.H.; Kalia, A.; Kaminsky, M.; Andersen, D.G.; Seongil, O.; Lee, S.; Dubey, P. Architecting to Achieve a Billion Requests per Second Throughput on a Single Key-Value Store Server Platform. In *ACM SIGARCH Computer Architecture News*; ACM: New York, NY, USA, 2015; Volume 43, pp. 476–488. [CrossRef]
31. Lim, H.; Han, D.; Andersen, D.G.; Kaminsky, M. MICA: A holistic approach to fast in-memory key-value storage. In Proceedings of the USENIX, Philadelphia, PA, USA, 19–20 June 2014.
32. Lim, K.; Meisner, D.; Saidi, A.G.; Ranganathan, P.; Wenisch, T.F. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *ACM SIGARCH Computer Architecture News*; ACM: New York, NY, USA, 2013; Volume 41, pp. 36–47. [CrossRef]
33. Andersen, D.G.; Franklin, J.; Kaminsky, M.; Phanishayee, A.; Tan, L.; Vasudevan, V. FAWN: A fast array of wimpy nodes. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, USA, 11–14 October 2009; ACM: New York, NY, USA, 2009; pp. 1–14. [CrossRef]
34. Dabek, F.; Kaashoek, M.F.; Karger, D.; Morris, R.; Stoica, I. Wide-Area Cooperative Storage with CFS. In *ACM SIGOPS Operating Systems Review*; ACM: New York, NY, USA, 2001; Volume 35, pp. 202–215. [CrossRef]
35. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. In *ACM SIGOPS oPerating Systems Review*; ACM: New York, NY, USA, 2007; Volume 41, pp. 205–220.

36. Singh, A.; Ong, J.; Agarwal, A.; Anderson, G.; Armistead, A.; Bannon, R.; Boving, S.; Desai, G.; Felderman, B.; Germano, P.; et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google'S Datacenter Network. In *ACM SIGCOMM Computer Communication Review*; ACM: New York, NY, USA, 2015; Volume 45, pp. 183–197. [CrossRef]
37. CAVIUM. XPliant Ethernet Switch Product Family. 2017. Available online: https://www.cavium.com/xpliant-ethernet-switch-product-family.html (accessed on 7 November 2018).
38. Ozdag, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. *See goo.gl/AnvOvX*, 2012; p. 5.
39. Sivaraman, A.; Cheung, A.; Budiu, M.; Kim, C.; Alizadeh, M.; Balakrishnan, H.; Varghese, G.; McKeown, N.; Licking, S. Packet transactions: High-level programming for line-rate switches. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; ACM: New York, NY, USA, 2016; pp. 15–28. [CrossRef]
40. Ghasemi, M.; Benson, T.; Rexford, J. Dapper: Data plane performance diagnosis of tcp. In Proceedings of the Symposium on SDN Research, Santa Clara, CA, USA, 3–4 April 2017; ACM: New York, NY, USA, 2017; pp. 61–74. [CrossRef]
41. Narayana, S.; Sivaraman, A.; Nathan, V.; Goyal, P.; Arun, V.; Alizadeh, M.; Jeyakumar, V.; Kim, C. Language-directed hardware design for network performance monitoring. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Los Angeles, CA, USA, 21–25 August 2017; ACM: New York, NY, USA, 2017; pp. 85–98. [CrossRef]
42. Dang, H.T.; Canini, M.; Pedone, F.; Soulé, R. *Paxos Made Switch-y*; ACM: New York, NY, USA, 2016; Volume 46, pp. 18–24. [CrossRef]
43. Sapio, A.; Abdelaziz, I.; Aldilaijan, A.; Canini, M.; Kalnis, P. In-Network Computation is a Dumb Idea Whose Time Has Come. In Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, 30 November–1 December 2017; ACM: New York, NY, USA, 2017; pp. 150–156. [CrossRef]