Enabling Efficient Multi-Type Load Awareness for In-Network Key-Value Stores

***Abstract*—Distributed key-value stores are critical to high-performance online services (*e.g.,* social networking and e-commerce) but face challenges of load imbalance. To efficiently balance loads, in-network key-value stores route, cache, and replicate key-value items based on load statistics. However, existing in-network solutions are either infeasible or inefficient to count *multi-type* loads (*e.g.,* content, storage, and network), whose enormous key space may quickly overwhelm the memory budgets and the control-plane bandwidth. In this paper, to efficiently be aware of multi-type loads, we (1) systematically identify and analyze the load types that may cause performance bottlenecks; (2) propose a framework called Lake, which enables efficient multi-type load awareness with stochastic variance minimization theory; (3) identify some correct-by-construction optimization techniques based on load statistics. Finally, based on our prototype implementation on Tofino switches, our extensive experiments indicate that Lake provides higher accuracy (*e.g.,* F1 score from 0.88 to 1) with acceptable hardware consumption (*e.g.,* up to 56.51% hash call reduction) for multi-type load awareness.**

I. Introduction

Modern Internet services such as e-commerce and social networking heavily rely on distributed key-value store systems to meet strict Service Level Objectives (SLOs, *e.g.,* 10-100ms response latency [1, 2]). In practice, a major challenge of key-value stores is to meet the strict SLOs under highly *skewed* and *dynamic* workloads, which may cause load imbalances and significantly reduce system throughputs.

To address the challenge, in-network key-value stores [3–5] are *aware* of and *balance* loads in real time by leveraging programmable switches. For example, NetCache [3] deploys a load-balancing cache on a Top-of-Rack (ToR) switch. The cache on a ToR switch increases system throughput at orders of magnitude due to switch visibility and high-performance switch ASICs (*i.e.,* >10Tbps ASIC[6]).

However, the existing in-network solutions are either infeasible or inefficient to comprehensively be aware of *multi-type* loads (*e.g.,* content, storage, and network) on current programmable switches. For *infeasibility*, they may fail to be aware of *multi-type* loads simultaneously on a programmable switch. For *inefficiency*, multi-type load awareness may consume a considerable amount of switch resources.

This limitation is fundamentally rooted in the enormous key space of loads, which may fast consume switch resources. Specifically, register-based [3–5, 7] works are aware of loads with register arrays. But the enormous key space of loads (2^{128} loads for 16B keys) may quickly overwhelm scarce register resources (NetCache [3] only supports 2 sketches in our benchmark analysis § III.B). The sample-based works [5] reduce the memory footprints but may lose significant loads due to the limited control plane (CP) bandwidth (up to 44.54% drop rate in our benchmark analysis § VI Exp#2).

Consequently, the systems may be blocked by unaware load types and suffer performance penalties. The load awareness may fail to co-exist with basic switch functionalities (*e.g.,* flow table entries) and in-network key-value functionalities (*e.g.,* cache or coherence directories). Thus, it is significant to explore if, and how, we can efficiently deploy multi-type load awareness on programmable switches. This is the key problem that this paper addresses.

In this paper, we identify three load types that potentially introduce bottlenecks on in-network key-value store systems:

- *Content workloads.* The requests from clients might introduce bottlenecks due to dynamical and rapidly-changed load distributions, *e.g.,* write-intensive requests may introduce significant consistency overheads [3, 5].
- *Network workloads.* The network traffic in switches might lead to bottlenecks due to network congestion. *e.g.,* the incast [8] may occur in a switch under the Memcached many-to-many network connection model [9].
- *Storage workloads.* The storage servers might be overloaded due to uneven item partitioning, *e.g.,* loads access is highly skewed so that the storage loads may exceed the processing capacity of a single server [5].

Having identified these load types, our contribution is a unified abstraction of multiple loads to enable efficient multi-type load awareness tailored for in-network key-value stores. Specifically, we *(1)* transform the awareness of multi-type loads into a subset-sum estimation problem to provide a unified abstraction of multi-type loads; *(2)* design a probabilistic framework called Lake based on the abstraction to enable multi-type load awareness with theoretical guarantees; *(3)* synthesize and identify some known and novel practical optimization techniques for load balancing based on the awareness results; *(4)* design a high-level language to express user requirements with hardware optimizations so that they can customize the load awareness efficiently. In practice, users can simply assemble load fields and be aware of multi-type loads by Lake with theoretical guarantees. The load statistics allow users to comprehensively identify bottlenecks and design load balance algorithms for multi-type loads.

In our evaluation, we conduct extensive experiments on Barefoot Tofino switches [10]. The results show that Lake accurately counts multi-type loads, (*e.g.,* 5 load types) with fewer resource footprints (*e.g.,* zero load drop reporting to the control plane), and enables efficient load-balancing techniques (*e.g.,* up to 4.03× MQPS).

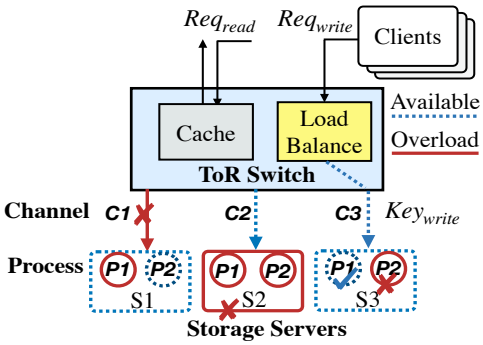


Fig. 1. **Key-value stores have the incentive to balance loads based on multi-type loads:** *a)* In-network cache cannot benefit write-intensive requests, which could be directly scheduled to servers to improve throughput; *b)* Requests could be scheduled to the process P1 in the server S1 but the channel C1 may block the requests; *c)* Requests could be scheduled via channel C2 but P1 in S1 may block the requests; *d)* Requests could be scheduled to process P1 in server S3 via channel C3 without blocking.

II. Background

In this section, we start with the background on workload distributions. Next, we introduce the emerging programmable switch, which inspires some promising architectures of in-network key-value stores.

A. Skewed Workloads in Key-Value Stores

When scaling distributed key-value stores to billions of clients, a major challenge is to meet the strict SLOs under highly *skewed* and *dynamic* workloads. Specifically, *(1)* for high skewness, popular items receive far more requests (millions per day) than others (almost none) [11, 12], and *(2)* for dynamic distributions, the set of popular items changes dynamically and rapidly become unpopular <10min [13]. The popular items may suddenly crowd into a few servers, leading to server overload and performance penalties [5].

The workloads of real-world key-value store systems are often modeled using Zipfian distributions [11, 12, 14, 15], typically exhibiting highly skewed patterns(*e.g.,* Zipf distributions with α>1 [12, 16]). The key-value store system partitions and distributes items to multiple servers to ensure SLOs. The skewed workloads mainly include skewed content loads, network loads, and storage loads. These skewed loads may overload a small portion of resources, which may become new bottlenecks and introduce longer tail latency. Over-provisioning resources may reduce performance penalty but significantly increases overall cost [5].

B. Programmable Switches

The emerging programmable switches provide customizable line-rate packet processing (>10Tbps [6]) in the data plane (DP). The typical Reconfigurable Match-Action Tables (RMT) paradigm [17] maintains a pipeline of stages, each of which is equipped with an identical design, the same amounts of resources (*e.g.,* SRAM and SALU). Each stage maintains a match table that matches the fields of packet headers to an assigned value and executes simple instructions by action units. The switch resources are very limited. For instance, a typical commercial switch maintains 12 pipeline stages with 10MB TCAM and SRAM, 10 SALUs, and 10 hash calls [17–19]. Users write a data-plane program in P4 language [20]. A P4 compiler maps the programs into the pipeline only when the required resources are sufficient.

Table I. The typical systems of in-network key-value stores

In-network System	Load Awareness	Load Balancing
SwitchKV [4]	Hotkeys	Route to storage servers
	Burst hotkeys	Adapt to new workloads
NetCache [3]	Cached keys	Update cache sets
	Uncached keys	Cache new hotkeys
	Remove duplicate	Remove the duplicate keys
Pegasus [5]	Server loads	Balance the loads of servers
	Hotkeys	Select hot replicas
	Write keys	Select hot replicas

C. In-network Key-Value Stores

In-network key-value store systems leverage emerging programmable switches to route, cache, and replicate popular items [3–5]. Typically, these systems are *aware* of multi-type loads due to the switch visibility (*i.e.,* request aggregation point) and *balance* the loads in real-time because of the high-performance switch ASICs (*i.e.,* >10Tbps [6]). As such, these systems can provide high throughput, low latency, and scalability for key-value stores. Next, we ellaborate on each typical work with two phases: *load awareness* and *load balancing*.

In-network cache. NetCache [3] takes the ToR (Top-of-Rack) switch as a load-balancing cache (medium cache hit ratio<50% [4]), which absorbs hot items and makes the loads on storage servers more uniform. In the *load-awareness* phase, it maintains a CM sketch that detects hot uncached keys and a Bloom filter that eliminates the replicated keys in the data plane. At the end of each time window, the statistics are reported to the control plane, where the controller determines the cache set. In the *load-balancing* phase, the cache sets in the data plane are updated by new ones. The cached keys directly reply read requests to clients while the write requests are forwarded to servers to ensure consistency.

Content-based routing. SwitchKV [4] divides storage nodes into slow storage nodes and in-memory cache nodes and maintains the addresses of each key so that it can fast route requests to the storage node. In the *load-awareness* phase, it keeps a local frequency counter for the recently visited keys in each backend storage node. For the keys whose frequency is greater than a threshold, SwitchKV periodically reports recent hotkeys and the keys that suddenly become very hot to the cache nodes. In the *load-balancing* phase, the in-memory cache nodes add or evict selected keys, and directly route the requests to the corresponding storage nodes.

In-network replication Pegasus [5] distributes the load by replicating the hot items to multiple storage servers. It rebalances item replicas while guaranteeing data consistency and coherence. The switch maintains an in-network coherence directory that tracks and manages the location of replicated items. In the *load-awareness* phase, it samples requests for unreplicated keys, forwarding them to the switch CPU, where a program counts the access frequency. Each server collects load statistics and reports to the controller, where two separate register arrays track the READ/WRITE count of each repli-

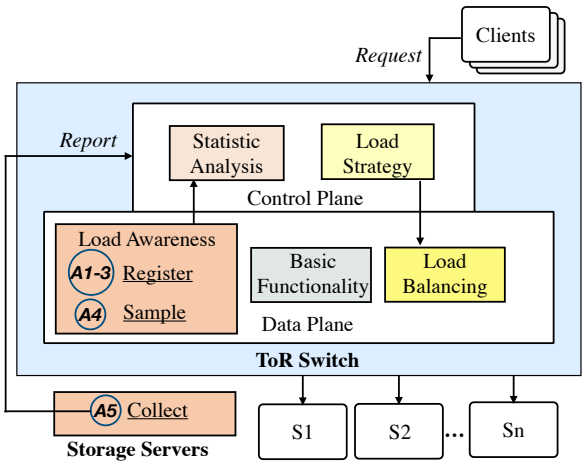


Fig. 2. The in-network architecture and load-awareness alternatives.

cated key. In the *load-balancing* phase, the controller picks replicas based on the load statistics.

III. Motivation

In this section, we elaborate on the motivation of this paper.

A. Why Multi-type Load Awareness

The workloads mainly include three types content, network, and storage. As shown in Fig. 1, any load type may introduce performance bottlenecks. So we should schedule multi-type loads comprehensively to address these bottlenecks. Moreover, the existing key-value store systems have the incentive to balance loads based on multi-type loads. For example, Pegasus is aware of write- and read-intensive requests and replicates both requests with different server selection policies. Because the cost of write-intensive requests can easily negate load-balancing benefits [5]. Furthermore, the existing system requires to balancing the loads at different granularities. For instance, The system [9] balances loads at a server level or a process level to improve the overall system utilization.

However, the existing systems are unable to efficiently be aware of multi-type loads. To analyze the root cause, we define a multi-type load, which puts the existing requirements together: L (*e.g.,* 184bit) = {key frequency (key_f, *e.g.,* 128bit keys), the write-key frequency (key_w, *e.g.,* 1bit flag), the physical addresses of storage server ($addr_{phy}$, *e.g.,* 32bit IP address), the virtual addresses of processes ($addr_{vir}$, *e.g.,* 16bit port number), loads of network channels (cha_{id}), *e.g.,* 7bit switch ports}. Next, we take L as an example to analyze the limitations of existing works and the potential alternatives.

B. Existing Solutions and Limitations

As shown in Table I, we compare the alternatives that provide multi-type load awareness. We consider 6 metrics: the accuracy of load awareness, load awareness speed, state consistency between the control plane (CP) and the data plane (DP), the number of types to be aware of, DP resource footprints, and the bandwidth usage between CP and DP.

Alternative#1 (A1): Multi-type load awareness with register arrays. An alternative is to count *multi-type* loads in the register arrays. But this alternative is expensive and even infeasible. For example, the key space may have 2^128 for 16B keys. However, the hardware resources are very limited(*e.g.,* 10MB SRAM [17, 19]). So the existing works [3, 5] can only count the cached/replicated keys with register arrays to reduce SRAM consumption. Consequently, under rapidly-changed workloads, the keys suddenly becoming popular may not be captured by the register arrays. The key-value stores may lose the hotkeys and suffer from performance penalties.

Table II. The comparation of trade-off of each alternative load awareness

Alternative/ Position	Accuracy/ Speed	Consistency/ Load	DP Resource	CP BW/ Query
A1/DP	High/High	Strong/**Partial**	**Infeasible**	Low/Fast
A2/DP	High/High	Strong/**Partial**	**Infeasible**	Low/Fast
A3/DP	**Low**/High	Strong/Full	Low	Low/**Slow**
A4/DP	**Low**/High	Strong/Full	Low	**High**/Fast
A5/Server	High/Low	**Weak**/Full	Low	**High**/Fast
Lake/DP	**High/High**	**Strong/Full**	**Low**	**Low/Fast**

Alternative#2 (A2): Each sketch for per load type. Sketch [21–23] is an efficient approximate data structure. Another alternative is to count each load using each sketch. For example, we can deploy 5 CM sketches, each of which counts one load in L with registers. However, a Tofino switch cannot support more than 4 sketches [24]. Worse, some key-value systems can only deploy 2 sketches (demonstrated in our benchmark analysis). Further, it must ensure the co-existence with basic switch functionality (*e.g.,* firewalls) and in-network key-value functionality (*e.g.,* cache or coherence directories). Thus, this alternative is either infeasible or inefficient.

Alternative#3 (A3): All load awareness with post-recovery. Another sketch-based alternative is to be aware of all loads (*i.e.,* L) simultaneously and recover partial loads (*e.g.,* $addr_{vir}$) by aggregating other loads. But related works [24, 25] have demonstrated that this approach may impose high estimation bias. Further, enormous key space may impose significant query overheads (*e.g.,* aggregating $2^{184}/2^{48} = 2^{136}$ keys for 2^{48} $addr_{vir}$ from $2^{184}bit$ L).

Alternative#4 (A4): Sample in switches and report to CP. An alternative is to sample load states and report to the CP. It reduces register consumption without storing the states. But it also introduces a new bottleneck: the switch PCIe has very limited bandwidth and may lose significant states. As a result, under a low sample rate, the alternative loses significant states (*e.g.,* the sudden change of workload distributions [4]). Under a high sample rate, the states may also be dropped by the CP due to the limited CP bandwidth (demonstrated in Exp#2).

Alternative#5 (A5): Collect in servers and report to CP. Another alternative is to collect the statistics of loads in each storage server and report these statistics to CP. The key limitation is the complex mechanism of consistency for each state. It should maintain state consistency and synchronization in each server. The network and processing capability may delay the transmission of these states. It is challenging to benefit the switch key-value stores in real time. Furthermore, it is hard for servers to be aware of switch network loads, which may indicate network congestion.

Benchmark analysis. In our benchmark analysis, we main-

tain the same configuration in NetCahce, *i.e.,* a CM sketch with 4 hash functions and 2^{16} 16bit registers and a Bloom filter with 3 registers arrays, each of which has 2^{18} 1bit registers. In a Tofino switch, we can only successfully compile the data-plane program with two CM sketches in our implementation, measuring 128bit key_f and 48bit $addr_{vir}$ (Exp#1 in detail).

Summary. In summary, the need for multi-type load awareness and the limitations of alternatives (A1-A5) motivate us to better be aware of multi-type loads.

IV. Multi-Type Load Awareness

In this section, we elaborate on the design goals and the challenges. Next, we illustrate how we transform the problem and propose a data structure to address these challenges.

A. Challenges

Goals. We aim to efficiently be aware of multi-type loads simultaneously without complex consistency mechanisms. Moreover, users can customize the load awareness flexibly.

Challenge#1 (C1): Versatile load awareness. It is challenging to support multi-type load awareness efficiently due the heterogenous load types. Specifically, different load types (*e.g.,* content, storage, and network) may have different significance and expression. As a framework, we should have an isomorphic expression for multi-type load awareness. Furthermore, the expression should be efficient under the strict constraints of switch hardware (addressed in § IV.B).

Challenge#2 (C2): Fidelity of heavy loads. The fidelity of heavy loads is critical because the load balancing may be misguided by inaccurate results. For example, we can evict the write-heavy items in the in-network cache because the write items cannot benefit from the in-network cache. But the read-heavy items mistakenly evicted may reduce the system throughput. Thus, we should ensure the fidelity of heavy loads. But the switch may fail to store all loads to ensure error-free awareness or fail to predict all the heavy flows due to the switch hardware constraints. Thus, it requires a careful design tailored for the fidelity of heavy flows (addressed in § IV.C).

Challenge#3 (C3): Metadata dependence. The metadata dependence of multi-type loads may consume significant pipeline stages. In the DP, the switch ASICs enable high-speed load awareness. To benefit from switch ASICs, it is inevitable to use register arrays for stateful awareness. To read or write the arrays, we should calculate the location of a key. But an array to update a metadata must wait for another one to finish metadata updating. Otherwise, multi-type loads must consume more stages of the switch pipeline to prevent data hazards, leading to excessive resource consumption or even compilation failures (addressed in § IV.F).

B. Multi-type Load Awareness

To provide a unified and formal abstract of load awareness, we define a new class of problem called *multi-type load awareness*, which supports multi-type load awareness simultaneously without the need for pre-defined loads. Specifically, we union all the loads to be aware of as a *full load* in which all *partial loads* can be recovered.

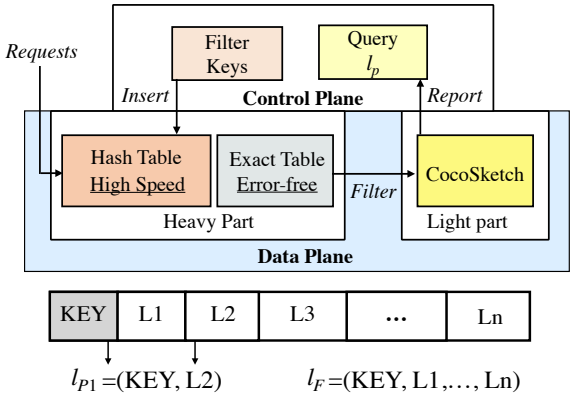


Fig. 4. The full load and a partial load.

Definition 1. *(Partial Load). A load l_P is a partial load of full load l_F (denoted by $l_P \prec l_F$), if there is a mapping $g(\cdot)$: $l_F \to l_P$, and for any key $e \in l_P$ defined on load l_P, we have $f(e) = \sum_{e' \in l_f, g(e')=e} f(e')$, where $f(e)$ is a statistic of e.*

Example. for a $l_F = (key, operations, server\_id)$ (*e.g.,* $(k_1, write, server_{31})$ means that the key_1 with a WRITE request is forwarded to $server_{31}$), the frequency of a key k_1 of a partial load (*e.g.,* $(k_1, write, *)$) equals the sum of the frequency of full-flow, $\sum_{i=0}^{31}(k_1, write, server_i)$. Note that a partial load can be any subset of loads in full loads, *e.g.,* $(key, *, *)$ is a partial load of the l_F.

Definition 2. *(Multi-Type Load Awareness). Given a full load and a metric function f, return the $f(e)$ of any key $e \in l_P$ for any partial load $l_P \prec l_F$.*

This definition allows users to only be aware of the l_F and can query any l_P at the end of each time window.

C. Load-Awareness Data Structure

Overview. As shown in Fig. 3, the architecture of Lake comprises a heavy part and a light part. To address C1, Lake *(1)* filters the heavy loads from the light part to ensure the fidelity of heavy loads, and *(2)* replaces the Bloom filter that removes the duplicate reports of heavy loads to reduce the consumption of control plane bandwidth. In the light part, we minimizing the stochastic variance of all the sub-set sum of L_F to reduce the estimation errors of all the l_P. Specifically, we use a CocoSketch in the light part, removing and reporting the heavy loads (*i.e.,* counts exceed a threshold) to the CP. Then, the heavy loads are inserted into the heavy part. For each load, the heavy part directly increments its counts in the data plane, bypassing the light part. Thus, the heavy part counts the heavy loads with negligible errors so that the fidelity of heavy load awareness is also improved. Moreover, the light part also reduces the hash collisions. Finally, the loads and their statistics are reported to the CP for further l_P query.

Data structure. As shown in Fig. 5, the data structure consists of two parts: a heavy part recording hot loads and a light part recording cold loads. The heavy part $\mathcal{H}$ contains

Table III. The notations of Lake.

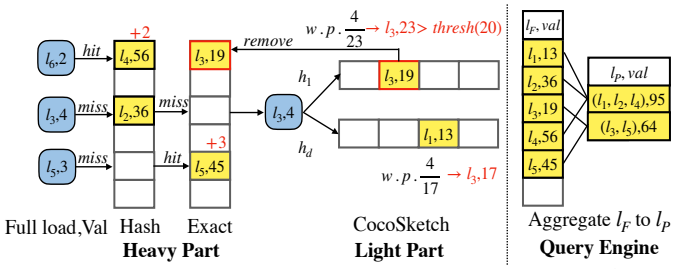
Symbol	Description
H	The hash table in the heavy part;
$h(.)$	The hash function of the hash table;
$H[i]$	The i^{th} bucket of hash table;
$H[i].K$	The key field in $H[i]$;
$H[i].V$	The value field in $H[i]$;
E	The exact table in the heavy part;
$E[i]$	The i^{th} bucket of the exact table;
$E[i].K$	The key field in $E[i]$;
$E[i].V$	The value field in $E[i]$;
C	The CocoSketch in the light part;
d	The number of arrays in CocoSketch;
l	The number of buckets in one array;
$h_i(.)$	The hash function of the i^{th} array in the light part;
$C_i[j]$	The j^{th} bucket in the i^{th} array in the light part;
$C_i[j].K$	The key field in $C_i[j]$;
$C_i[j].V$	The value field in $C_i[j]$;
$P()$	Probability substitution function;

a high-speed hash table and an error-free exact table. Each bucket of the heavy part records the counts of a load: load key and load value. The light part is a sketch that supports multi-type loads query (*e.g.,* CocoSketch). A Cocosketch maintains d arrays of l key-value pairs. Each array is associated with an independent hash function.

Lake insertion. When a load with a load key and load value needs to be inserted, the Lake extracts the load key and computes the hash functions to locate a counter in a hash table. When the load key is the same as the recorded key, Lake directly increases the load value to the count value. When the load key is different, Lake checks whether the corresponding recorded key in the exact table is the same. If the keys are the same, Lake increases the value of the corresponding counter. When the two tables of the heavy part are missed, the load will be recorded in the light part. CocoSketch first maps the load key to d buckets (each bucket comes from one of the d arrays) and updates each bucket independently, then replaces the recorded key with a $\frac{load\ value}{total\ value}$ probability. When the record value exceeds the threshold, Lake removes the load exceeding the threshold and reports it to the CP, and inserts the load into the heavy part through the CP.

Example. As shown in Fig. 4, the Lake counts the full loads. For the heavy load stored in the hash table, l_6, Lake directly increments its counter. For the heavy load that collided in the hash table but stored in the exact table, l_5, Lake also increments its counter. The heavy part can filter heavy loads fast and accurately. For the light loads l_1 and l_3, they are counted in the CocoSketch (the hardware version) while they benefit from the *Stochastic Variance Minimization* that can improve the accuracy of l_P awareness. The CocoSketch increments all the values and replaces the loads with a probability $\frac{value}{total\ value}$. At the end of each time window, the full loads are reported to the CP to construct a full load table. When querying partial loads (l_3, l_5), the Lake aggregates the full loads l_3 and l_5 and produces the final result 64.

Lake query. In the CP, Lake builds the load recorded by the heavy part and light part into a full load table containing two columns (load key, load value). When querying, the value of the partial load is queried through the aggregation of l_F.



Step 1: Filter hot loads and measure cold loads in DP **Step 2**: Query l_P in CP

Fig. 5. The detailed process of load awareness using Lake.

D. Theory Analysis

Due to the separation of hot and cold loads, Lake has high accuracy in most cases. Before the analysis, we will define the following symbols. We denote the real value of load e by $l(e)$ and its estimated value by $\widehat{l}(e)$, and each new load and its value by (e_i, w).

Stochastic Variance Minimization. We first analyze the minimization of the variance sum of loads. Our ultimate target is the minimization of load sum estimation.

$$\text{minimize} \sum_e (l(e) - \widehat{l}(e))^2 \tag{1}$$

In the light part, we need to consider the incremental minimization of the sum of variance caused by a load of each insertion.

$$\text{minimize} \sum_e \Delta(l(e) - \widehat{l}(e))^2 \tag{2}$$

We ensure that the variance change caused by each insertion is minimized so that Lake can reduce the overall estimation errors of multi-type loads.

Lemma 1. *The minimum increment of variance sum to update the bucket* (e_j, l_j) *is*

$$\sum_e \Delta(l(e) - \widehat{l}(e))^2 = \begin{cases} 2wl_j, & e_i \neq e_j \\ 0, & e_i = e_j \end{cases} \tag{3}$$

Proof. For the heavy part, each insertion in the heavy part does not change the variance sum due to its error-free statistics. Therefore, each insertion of Lake can be guaranteed to minimize the variance sum increment. □

Error Bound. Subsequently, we discuss the estimation error of Lake. Consider a CocoSketch of $d * l$ in the light part. We use $R(e)$ to denote the relative error for load e. The following theorem proves the error bound of $R(e)$ in the light part.

Lemma 2. *Let* $l = 3 \cdot \epsilon^{-2}$ *and* $d = O\left(\log \delta^{-1}\right)$. *For any load e of multi-type load* $l_P < l_F$,

$$\mathbb{P}\left[R(e) \geqslant \epsilon \cdot \sqrt{\frac{f(\bar{e})}{f(e)}}\right] \leqslant \delta \tag{4}$$

Proof. For hot loads stored in the heavy part, errors exist only due to hash collisions in the light part before insertion into the heavy part. Since the lightweight part is dominated by cold loads, there are fewer hash collisions and fewer errors are generated. At the same time, the value in the lightweight part accounts for a small proportion of the hot loads, so the error of the hot loads is negligible. In the worst case, that is, a bucket in the heavy part is not updated after insertion. The error bound of $R(e)$ can still be obtained by theorem 2. □

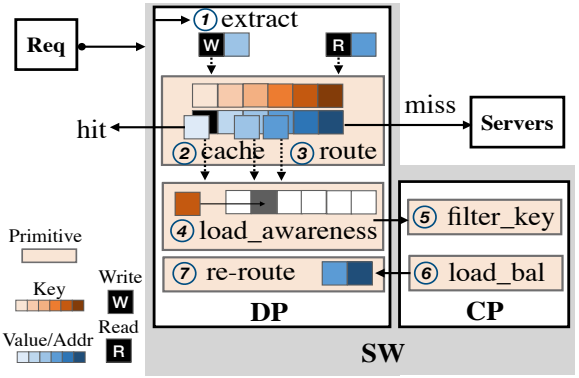


Fig. 6. Putting O1-O4 together with Lake.

Theorem 1. *Let* $l = 3 \cdot \epsilon^{-2}$ *and* $d = O\left(\log \delta^{-1}\right)$. *For any load* e *of multi-type loads* $k_P < k_F$,

$$\mathbb{P}\left[R(e) \geqslant \epsilon \cdot \sqrt{\frac{f(\bar{e})}{f(e)}}\right] < \delta \tag{5}$$

In summary, we can conclude that the error bound of $R(e)$ of Lake can be obtained by Theorem 3.

E. Query-based Load Awareness

We provide a front-end language for users to express their requirements for load awareness. The language is translated into P4 codes while benefiting from hardware optimization.

Query language. In the control plane, we first build a table with two columns (Full Load, Size) by querying the Lake on the recorded full-key flows. For the heavy part, we directly place the full load and its value on the table. For the light part, we take the median estimated size in different arrays as its final estimated size like the original hardware version [24]. We define SQL statement interface to query the awareness results:

```
SELECT g(l_F), SUM(Value)
FROM Table_light A
INNER JOIN Table_heavy B
ON A.Key = B.Key
GROUP BY g(l_F)
```

where g(l_F) is the mapping from l_f to l_p.

F. Hardware Optimization

While Lake can accurately count multi-type loads with stochastic variance minimization and heavy loads separation, it may not be efficient or successfully deployed for key-value stores when implemented in hardware platforms. To address the C3, we remove the metadata dependence and provide a principle to aggregate the heavy part.

Metadata dependence. In key-value stores, the key length may exceed the maximum bit width of a register array. To store the key, we split a key into slices and use multiple register arrays to store each slice respectively. However, the multiple register arrays may depend on one metadata to index the location of a key in the array. As a result, a register array must wait for another one to process the metadata. Multiple register array depends on one metadata so that it consumes a number of pipeline stages. For example, to count a 184bit full load (*e.g.,* L) in the light part, we use one 32bit register array to count values, 4 32bit register arrays to store keys, and a 1bit, 7bit, 16bit, 32bit register array to store WRITE/READ types, the channel ID, ports, IP addresses respectively. Moreover, in the heavy part (hash table), we should also use 9 register arrays to store the full load. These register arrays rely on the hash function to previously calculate the index of the key, leading to multiple register arrays depending on one metadata and consuming a large number of stages.

Algorithm 1 Lake Insertion Algorithm

Input: key: load key; $value$: load value; H; E; C;
Output:

```
if H[h(key)].K == key then
    H[h(key)].V = H[h(key)].V + value
else if E[h(key)].K == key then
    E[h(key)].V = E[h(key)].V + value
else
    for i=0; i<d; i++ do
        val tmp = h_i(key)
        C_i[tmp].V = C_i[tmp].V + value
        val probability = value / C_i[tmp].V
        C_i[tmp].K = P(C_i[tmp]).K, key, probability)
    end for
end if
```

Parallelize index calculation. To remove the metadata dependence, we *parallelize* the calculation of the index due to the architecture of RMT. Specifically, we calculate multiple different indexes in parallel with multiple hash functions so that each register array can obtain the index without dependence on one metadata. Although the parallelism consumes a handful of metadata values, it is negligible compared with the total budget of SRAM blocks (*e.g.,* 10MB). We demonstrate these in our evaluation (exp#1).

Heavy part aggregation. The hash table in the heavy part filters the heavy loads while improving the accuracy of the light part. The hash table is efficient for short keys. For longer keys, the hash table may split the key into slices to store and hash the slice, and compare whether the key is identical to the one stored in the hash table. But we observe that we can sacrifice a little control-plane bandwidth by aggregating the hash table into the exact table. Specifically, for each heavy load, we report to the control plane and insert it into the exact table. The consumption of control-plane bandwidth is negligible because we report a load to the control plane only when the load is first detected as a heavy load.

Parameter Configuration Users may have different resource budgets and SLOs, they can configure Lake with SketchGuide [19] to maximize SLOs given resource budgets.

V. Load Banlancing with Lake

Lake enables efficient multi-type load awareness for in-network key-value stores. based on the load statistics. In this section, we synthesize and identify 4 optimization techniques as shown in Table iii. Our goal here is not to propose an

Table IV. Lake Optimization Comparation

No	Load Type	Load awareness	Load Balancing	Thresh	Primitive	Applicability
O1	Network	Available bandwidth	Schedule the underutilized channels firstly	T_c	`load-bal`	cache/route/replication
O2	Storage	Server loads	Schedule the underutilized servers firstly	T_s	`re-route`	route/replication
O3	Storage	Process loads	Schedule the underutilized process firstly	T_p	`re-route`	route/replication
O4	Content	Write-intensive items	Eliminate write-intensive items in cache	T_w	`filter_key`	cache/route

optimal algorithm for load balancing but to show that the load balancing can benefit from Lake with simple scheduling. For 3 types of loads, we leverage these results to address the performance bottlenecks. Moreover, we define the primitive of each technique, *i.e.*, the basic and automatic P4 code. Due to the space limit, we list the technique applicability in Table IV.

A. Network Loads

With network load awareness, we count the channel loads and the used channel of each item.

Optimization 1 (O1): Scheduling the underutilized channels first to bypass network congestion. As network congestion may reduce the system throughput, we count the requests through each channel and schedule the requests to underutilized channels to bypass congested channels. Specifically, a channel whose access times exceed a threshold T_c is identified to be overloaded. Then, we schedule requests to those underutilized channels (primitive `load-bal`).

B. Storage Loads

With storage load awareness, we count both server and process loads, and re-route requests to maximize parallelism.

Optimization 2 (O2): Scheduling the underutilized servers first to balance the storage loads. We count server loads and schedule the requests to the underutilized ones. Specifically, a server whose access frequency exceeds a threshold T_s is identified to be overloaded. We re-route requests to the underutilized ones (primitive `re-route`). The keys may have multiple replications in storage servers, O2 only modifies the priority of server access to ensure correctness.

Optimization 3 (O3): Scheduling the underutilized process first to balance the storage loads. Similar to O2, we count and schedule the storage loads but the storage node is a process. Specifically, the threshold is noted as T_p and the primitive is `re-route`. O3 only modifies the priority of process access to ensure correctness.

C. Content Loads

In content load awareness, we count the write-heavy items.

Optimization 4 (O4): Eliminating write-intensive items in a cache to count more read-intensive items. A limitation of an in-network cache is that it cannot improve the throughput of write items. Worse, the write-intensive items may occupy the cache and reduce the opportunity to cache read-intensive items. Thus, we can count the write items whose frequency exceeds a threshold T_w and filter them in the in-network cache (primitive `filter_key`). The write-intensive items will trigger the cache miss and be directly forwarded to the servers. The write-intensive items would be lost under switch failures, but these write-intensive requests are directly forwarded to the servers without a mechanism to ensure consistency.

D. Putting O1-O4 Together

To provide hardware optimization, we identify the primitives so that we can eliminate the repetitive codes and reduce the consumption of hardware resources. For example, we put O1 to O4 together, we first ❶ extract the requests from clients (`extract`), then ❷ cache (`cache`) or ❸ route (`route`) these requests. Then, we ❹ are aware of multi-type loads using Lake (`load_awareness`) and ❺ filter the write-intensive keys from a cache in the control plane (O4, `filter_key`). Then, the controller ❻ balances the loads based on the awareness results (O1, `load_bal`). Then, we ❼ schedule the items by re-routing these items to the underutilized servers (O2, (`re-route`)) or processes (O3, `re-route`).

VI. Implementation and Evaluation

We implement and evaluate Lake to indicate that:

- Lake counts multi-type loads with acceptable hardware resource footprints in a Tofino switch (Exp#1).
- Lake counts multi-type loads with negligible control plane communication overheads (Exp#2).
- Lake exhibits high scalability as the number of requests increases with the same resource provisions (Exp#3).
- Lake exhibits higher scalability as the number of loads increases given the same resource provisions (Exp#4).
- Lake counts multi-type loads accurately than the state-of-the-art works given the same resource provisions (Exp#5).
- Lake exhibits high robustness under different workload skewness with the same resource provisions (Exp#6).
- Lake exhibits high robustness under different write ratios with the same resource provisions (Exp#7).
- Lake counts multi-type loads and identifies bottlenecks with acceptable estimation errors (Exp#8).
- Lake improves the system throughput and reduces the resource footprints with O1 to O4 (Exp#9).

A. Methodology

Testbeds. Our testbed comprises two server machines and one 6.5Tbps Barefoot Tofino switch. Each server machine is equipped with a 20-core CPU (Intel Xeon Silver 4210R) and 64 GB total memory. Both machines are equipped with one 100G NIC (Mellanox MT27800). One machine generates key-value queries as a client and sends requests to another one as a key-value storage server.

Workloads. We use skewed workloads following Zipf distribution with different skewness parameters (i.e., 0.8, 0.85, 0.9, 0.95, 0.99), which are typical workloads for testing key-value stores [4, 26] and are evidenced by real-world deployments [1, 27]. We use Zipf 0.99 in most experiments to demonstrate that Lake provides high accuracy even under extreme scenarios. We also show that Lake still provides low

Table V. (Exp#1/#2) Hardware resource utilization of Lake, which count 5 loads and mostly reduces resource consumption than NetCache measuring hotkeys (NC1), NetCache measuring hotkeys and hot processes (NC2). We list the resource reduction with NC1 (R1) and NC2 (R2).

Resource Type	NC1(%)	NC2(%)	LA(%)	R1(%)	R2(%)
Exact Match xbar	9.83	11.39	6.38	35.10	43.99
Hash Bits	5.37	7.93	4.81	10.43	39.34
Hash Dist Unit	20.83	31.94	13.89	33.32	56.51
SRAM	5.21	8.96	5.52	-5.95	38.39
Map RAM	7.81	14.06	7.12	8.83	49.36
TCAM	0.00	0.00	0.00	0.00	0.00
Meter ALU	14.58	22.92	22.92	-57.20	0.00
Stats ALU	0.00	0.00	0.00	0.00	0.00
Stages	58.33	91.67	41.67	28.56	54.54

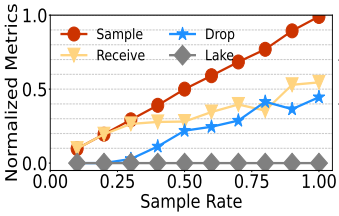


Fig. 7. (Exp#2) Control plane received fewer packets due to the bottlenecks of communication bandwidth.

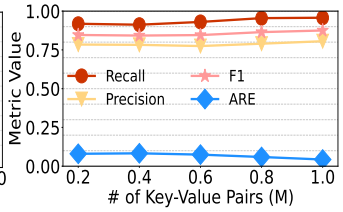


Fig. 8. (Exp#3) Lake maintains low estimation errors under the different number of requests.

estimation errors under less-skewed workloads (*e.g.,* Zipf 0.8). We use approximation techniques to quickly generate queries under a Zipf distribution in our clients [4, 28]. We use 16-byte keys and 128-byte values. The keyspace is hash partitioned across storage servers [29]. The write ratio ranges from 0.1 to 0.9. To count multi-type load, we are aware of the same L in section III.A.

Metrics. We are aware of the L with the following accuracy metrics, including recall rate $RR = \frac{TP}{TP+FN}$, precision rate $PR = \frac{TP}{TP+FP}$, F1-score, $F1 = \frac{2 \cdot P \cdot R}{P+R}$ $\frac{1}{|\Psi|}$, and average relative error (ARE) $\sum_{e \in \Psi} \frac{|f(e)-\hat{f}(e)|}{f(e)}$.

The hardware resources are scarce. Once the resource footprint exceeds the total resources, the program cannot be compiled successfully.

- SRAM. Stateful memory tracks/counts using tables/arrays, both of which consume SRAM blocks.
- SALU. The tables/arrays read/write the content or maintains hash calls, both of which consume SALUs.
- Hash call. The keys are hashed into specific counters, consuming pre-allocated hash call resources.
- Stages. The P4 program is compiled and allocated on the switch pipelines with limited resources.

VII. Testbed Experiments

We implement Lake in a Tofino switch [10] and have a 500KB memory budget. The heavy part is optimized using the techniques in section IV.F. The exact table has 1280 entries. The residual memory budget is all consumed by the CocoSketch in the light part.

Exp#1: Hardware resources. We show that Lake is able to count multi-type loads efficiently. We take the awareness part of NetCahce as the baseline, that is, the CM sketch and the Bloom filter, and eliminate other functions. We follow the same configuration in the original paper. For each load, we

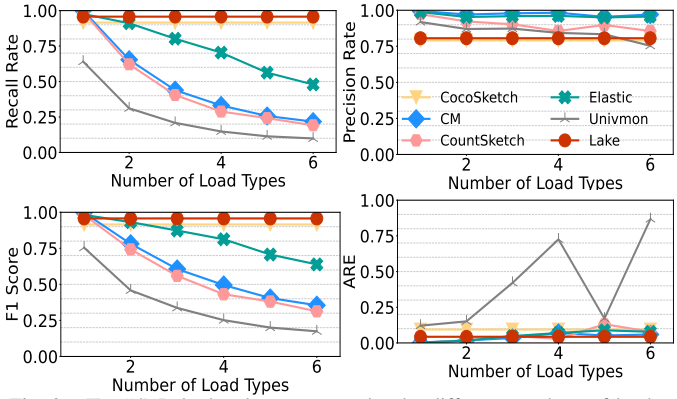


Fig. 9. (Exp#4) Lake has low errors under the different numbers of loads.

increment a CM sketch with the same configuration (following the A1). As shown in Table V, we count two loads: (1) the hotkeys (NC1) and (2) the hotkeys and the $addr_{phy}$ (NC2). NC2 consumes expensive 91.67% stages and fails to compile 3 loads (key_w). While Lake counts all 5 loads while consuming a proportion of hardware resources. Moreover, Lake mostly reduces the resource consumption significantly except SRAM and Meter ALU than NC1. But Lake does not consume more resources than NC2 while measuring 3 more loads. Lake achieves resource efficiency due to stochastic variance minimization, which reduces the errors of all loads.

Exp#2: Communication overheads. We demonstrate that A4 in section III may be limited by the tradeoff between the control plane bandwidth and the awareness accuracy. We send 1M packets by Pktgen [30] with a 10K rate. The data plane of the Tofino switch samples packets and reports to the switch OS. Fig. 7 shows that as the sample rate increases, the packets dropped by switch OS increases due to the limited control plane bandwidth. The maximum packets received by the switch OS is about 50%. It means that only about 50% packets would benefit the awareness results. Lake maintains a low drop rate because the switch OS only reads the Lake registers, which consumes very limited bandwidth.

A. Lake Scalability

We count the different number of loads given different memory budgets to exhibit the scalability of Lake. We compare with some state-of-the-art sketches as baselines, including CocoSketch [24], Count-Min sketch [23], Count sketch [31], Elastic sketch [21], and UnivMon [22]. We run Lake and these sketches in a python-based simulator, which is behaviorally equivalent to the data-plane program. The simulator allows us to freely run many accuracy experiments in different configurations. Each sketch has a 500KB memory budget and maintains the same configuration in the prior work [24]. We set the threshold to be 10^{-4} of the total counts of loads.

Exp#3: Lake scalability. To test the scalability of Lake, we change the number of requests. As the number of requests increases, the estimation errors keep stable even and slightly reduce. Lake is scalable because the heavy part prevents multi-type loads share the same counter so that it can accurately count the heavy loads. Moreover, the heavy part also increases

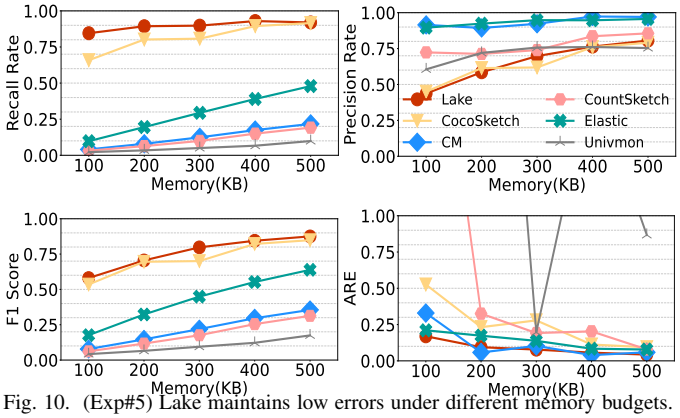


Fig. 10. (Exp#5) Lake maintains low errors under different memory budgets.

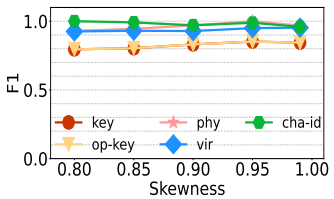


Fig. 11. (Exp#6) Lake maintains low error under different load skewness.

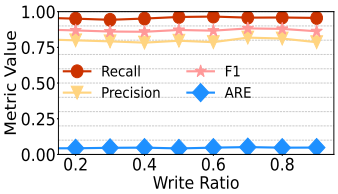


Fig. 12. (Exp#7) Lake maintains low error under different write ratios.

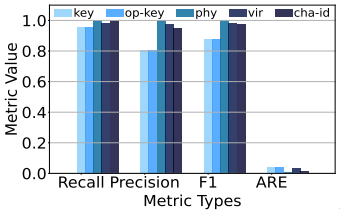


Fig. 13. (Exp#8) Lake counts different loads with low error

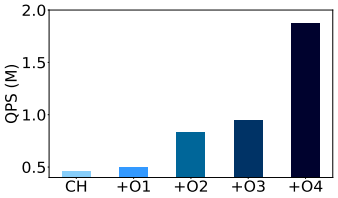


Fig. 14. (Exp#9) Lake improves the system throughput with O1-O4.

the accuracy when the number of key-value pairs increases because the loads could be filtered to the heavy part when the counts of heavy loads increase.

Exp#4: multi-type load scalability. As shown in Fig. 7, we gradually add new loads for each sketch. As the number of loads increases, some single-key sketches (*i.e.,* CM, Count sketch, Elastic sketch, and UnivMon) suffer from performance penalties. Although they have a higher precision rate than other sketches, they hardly identify hot loads (poor recall rate). Lake and Cocosketch maintain high scalability in all the accuracy metrics. Lake always has higher performance than CocoSketch because Lake filters the heavy flows from the light part so that both the heavy loads and the light loads can be accurately identified. Therefore, the in-network key-value store system could make more precise decisions due to the non-trivial identification of hot loads.

Exp#5: Memory scalability. We change the memory budgets and keep other setups the same as Exp#3. As shown in Fig. 9, Lake is more accurate than other sketches in the same memory budgets. Moreover, Lake has lower ARE than CocoSketch because the heavy part filters the heavy loads from the light part two to ensure the accuracy of the heavy loads.

B. Workload Awareness

Exp#6: Workload skewness. We count 5 different loads to test the robustness of Lake when the skewness of requests changes from 0.8 to 1. The F1 keeps stable with slight gains. As the skewness increases, the proportion of heavy loads increases and is shifted to the heavy part. Thus, Lake is robust when the workload distribution is highly skewed.

Exp#7: Write ratio. We count each metric value when the write ratio ranges from 0.2 to 0.9. Each metric value keeps stable, exhibiting high robustness in read-intensive workloads and write-intensive workloads. When querying the write-intensive workloads, Lake aggregates the partial loads with the same operations. The identification of operations is only 1 bit but Lake keeps high accuracy due to stochastic variance minimization, which customizes for subset sum estimation.

C. Load Balancing

Exp#8: Load awarenesss. We count 5 loads using Lake. Overall, Lake keeps high accuracy but also performs differently for different loads. The reason behind the difference is that a load with large key spaces may consume more memory footprints and imposes more hash collisions. Moreover, Lake filters the heavy loads from the light part so that it ensures high accuracy for heavy loads.

Exp#9: Load balancing. We demonstrate that simple optimization techniques can benefit from the identified optimization techniques (O1-O4) for in-network key-value stores. We use consistent hash (CH) as the baseline, gradually appending optimization techniques on the system (*e.g.,* "+O3" means we deploy O1, O2, and O3 on the system). With multi-type load awareness, we can deploy multiple optimization techniques under multiple constraints.

VIII. Related Works

A. In-network Load Awareness

In-network key-value stores are promising to improve load balancing under highly skewed and dynamic workloads. The representative works include in-network cache (*i.e.,* NetCache [3] and DistCache [7]), routing (*i.e.,* SwitchKV [4] and AppSwitch [32]), and replication (*i.e.,* Pegasus [5] and KVSwitch [33]). However, these works may fail to efficiently or even infeasible to count multi-type loads (demonstrated in section III), let alone to address the performance bottlenecks in each load. Lake enables efficient multi-type load awareness, providing the opportunity to address these bottlenecks.

B. Load Balancing

Traditional methods such as consistent hashing [29] and virtual nodes [29] but fall short under changing workloads [3]. "Power of two choices" [34] and data migration strategies [9, 35] can balance dynamic workloads but introduce additional system overheads or hard-to-handle highly skewed workloads [3]. Lake enables efficient multi-type load awareness so that the key-value system can efficiently optimize load balancing in the network with simple techniques.

IX. Conclusion and Future Works

In this paper, we propose a framework called Lake to enable efficient multi-type load awareness for in-network key-value stores. Lake consumes acceptable resources with high accuracy. Under different workload distributions, Lake

is highly scalable and robust. Based on the load statistics, the key-value stores can fast identify and address bottlenecks. The framework allows users to customize for load awareness and freely schedule the workloads. In the future, we plan to scale Lake to multiple racks and design more optimization techniques for in-network key-value stores.

REFERENCES

[1] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, 2017.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[3] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proc. of ACM SIGOPS SOSP*, 2017.

[4] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with $switchkv$," in *Proc. of NSDI'16*, 2016.

[5] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports, "Pegasus: Tolerating skewed workloads in distributed storage with $in-network$ coherence directories," in *Proc. of OSDI'20*, 2020.

[6] (2022) Barefoot tofino 2: Second-generation of world's fastest p4-programmable ethernet switch asics. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/

[7] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "Distcache: Provable load balancing for $large-scale$ storage systems with distributed caching," in *Proc. of FAST'19*.

[8] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and analysis of tcp throughput collapse in cluster-based storage systems." in *FAST*, 2008.

[9] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. of European Conference on Computer Systems*, 2015.

[10] (2022) Barefoot tofino. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. of SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012.

[12] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *Proc. of OSDI'20*, 2020.

[13] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter *et al.*, "The $cachelib$ caching engine: Design and experiences at scale," in *Proc. of OSDI'20*, 2020, pp. 753–768.

[14] ——, "The $cachelib$ caching engine: Design and experiences at scale," in *Proc. of OSDI'20*, 2020, pp. 753–768.

[15] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proc. of ACM SoCC*, 2010.

[16] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "$hotring$: A hotspot-aware in-memory key-value store," in *Proc. of FAST'20*, 2020.

[17] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM CCR*, 2013.

[18] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "Sketchlib: Enabling efficient sketch-based monitoring on programmable switches," in *Proc. of NSDI*, 2022.

[19] Z. Zhou, J. Lv, L. Cheng, X. Chen, T. Zhang, Q. Huang, J. Luo, L. Zhu, D. Zhang, and C. Wu, "Sketchguide: Reconfiguring sketch-based measurement on programmable switches," in *Proc. of ICNP*. IEEE, 2022.

[20] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, 2014.

[21] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. of ACM SIGCOMM'18*, 2018.

[22] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. of ACM SIGCOMM'16*, 2016.

[23] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, 2005.

[24] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "Cocosketch: high-performance sketch-based measurement over arbitrary partial key query," in *Proc. of ACM SIGCOMM'21*, 2021.

[25] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *Proc. of International Conference on Management of Data*, 2018, pp. 1129–1140.

[26] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding." in *Proc. of OSDI*, 2016.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of ACM SoCC*, 2010.

[28] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *Proc. of SIGMOD'94*, 1994.

[29] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. of the twenty-ninth annual ACM symposium on Theory of computing*, 1997.

[30] Pktgen. [Online]. Available: https://github.com/danielltt/pktgen

[31] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002.

[32] E. Cidon, S. Choi, S. Katti, and N. McKeown, "Appswitch: Application-layer load balancing within a software switch," in *Proc. of the First Asia-Pacific Workshop on Networking*, 2017.

[33] Y. Shi, J. Fei, M. Wenz, and C. Zhang, "Kvswitch: An in-network load balancer for key-value stores," in *Proc. of IEEE ISCC*, 2019.

[34] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[35] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper, "The yahoo! cloud datastore load balancer," in *Proc. of the fourth international workshop on Cloud data management*, 2012, pp. 33–40.