SpotMon: Enabling General Hotspot Monitoring in Key-Value Stores

Zhengyan Zhou[†*], Jinhan Zu[†], Enhao Huang[†], Zizhao Wang[†],
Haifeng Zhou[¶], Dong Zhang[*§‡], Xiang Chen[†], Chunming Wu[†*]
College of Computer Science and Technology, Zhejiang University[†], Quan Cheng Laboratory[*],
College of Control Science and Engineering, Zhejiang University[¶],
College of Computer and Data Science, Fuzhou University[§], Zhicheng College, Fuzhou University[‡]

***Abstract*—Key-value stores are essential to online services such as e-commerce. In key-value stores, a *hotspot* (*i.e.*, frequently accessed items) may cause severe load imbalances, high response latency, and Service Level Agreement (SLA) violations. However, existing works only focus on specific types of hotspots, thus overlooking other types of hotspots and leading to blind spots. In this paper, we propose *SpotMon*, a system that enables general hotspot monitoring in key-value stores. Specifically, we (1) systematically identify the generality requirements of hotspot monitoring from existing works, (2) formulate general hotspot monitoring as an arbitrary partial spot query problem, (3) measure the hotness of hotspot candidates with a new vector expression, (4) propose hotspot encoding, filtering, decoding, and querying to support general queries without focusing on specific hotspots, (5) leverage the in-network visibility of programmable switches to identify system-wide hotspots. Our extensive experiments indicate that SpotMon provides high accuracy (*e.g.,* F1 score from 0.88 to 1) and enables efficient hotspot mitigations (*e.g.*, up to 4.03× MQPS).**

I. Introduction

Key-value stores are building blocks in the storage infrastructure of services such as e-commerce. These services typically store items in a *key-value* format. Usually, users fetch items with three steps: (1) users send requests with *keys* to the services; (2) the storage system finds items according to the *keys*; (3) the system sends the *values* to users. Key-value stores have been widely adopted in the production of storage infrastructures, such as Redis [1] and Memcached [2].

In key-value stores, hotspots refer to a small set of popular items accessed disproportionately more than others. For example, breaking news could attract 90% access to 1% of items [3]. Hotspot monitoring is critical due to the severe influence of hotspots. For instance, every 500ms of additional loading delay in Google search would lose 20% of traffic [4].

Existing works craft special-purpose *monitoring approaches* tailored to specific *mitigation strategies* [3, 5–21]. For instance, Minos [10] mitigates head-of-line blocking by distributing large-size requests to independent NIC queues. To achieve this, it only monitors large-size hotspots in NICs. Such a special-purpose approach is compact and precise.

However, the monitoring approaches and mitigation strategies are tightly coupled in these works. As a result, these works only monitor specific hotspots, while other significant hotspots may be overlooked. These hotspots may become blind spots and may fail to be mitigated. This couple means that monitoring must be customized for mitigations. As such, hotspots not considered in advance will become blind spots.

In this paper, we pose a fundamental research question: *Can we break this couple and support general hotspot monitoring for key-value stores?* To determine the generality level, we systematically survey existing works (please refer to Table 1). First, they identify the workload types of a hotspot, including contents, operations, and size. Second, they need to locate a hotspot, such as servers, channels, and switches. In summary, we aim to allow querying the count value of a hotkey, workload types, overload locations, and any combination of these. This generality definition does not assume any specific hotspot to be monitored. On the flip side, it allows mitigation strategies to "late bind" to hotspots monitored in practice.

Nevertheless, it is prohibitively expensive to support general monitoring due to the non-trivial number of hotspot candidates. More precisely, it needs enormous key space to cover any combination of a key, workload types, and overload locations. For instance, 16-byte keys and 2^{32} IP addresses consume 2^{160} counters (more details in section § II).

To address this challenge, we propose *SpotMon*, a system that supports general hotspot monitoring by breaking the tight couple between monitoring and mitigations. The core idea of SpotMon is taking general hotspot monitoring as an arbitrary partial key query [22], *i.e.*, counting any partial combination of keys, workload types, and overload locations. Specifically, SpotMon (1) expresses keys, types, and locations with a hierarchical encoding technique; (2) measures item hotness with a new vector expression hotspot skyline; (3) maintains hotspot candidates in an approximate data structure (*i.e.*, Sketch) according to the hotspot skyline; (4) decodes and queries each hotspot with user interfaces.

We further implement SpotMon on a programmable switch. As such, SpotMon can observe all requests, types, and locations. Moreover, its high packet-processing rate (*i.e.,* >6.5Tbps [23]) provides real-time monitoring. Our contributions include:

- We identify the tight couple between monitoring and mitigations in key-value stores. We identify the requirement of general hotspot monitoring from existing works (§ II).
- We propose SpotMon that breaks the tight couple with hierarchical encoding, a hotness vector expression, and a sketch maintaining hotspot candidates (§ IV).

979-8-3503-5171-2/24/$31.00 ©2024 IEEE

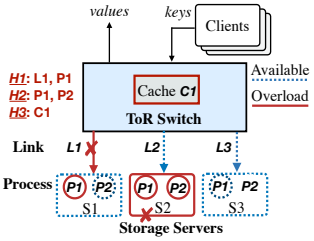

Fig. 1. **Benchmark Analysis (i)**: Special-purpose hotspot monitoring only focuses on both H1 and H2. Without mitigating H3, the system throughput is only slightly improved.

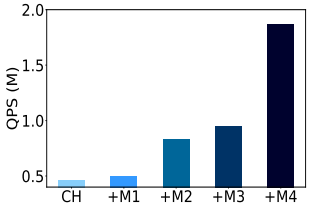

Fig. 2. **Benchmark Analysis (ii)**: M1, M2, and M3 only focus on H1 and H2, missing the opportunity of monitoring H3. Mitigating H3 significantly improves system throughput.

- We implement and evaluate SpotMon on a Tofino switch with extensive experiments. We show that SpotMon supports accurate general hotspot monitoring (0.88-1 F1 score) and further enables efficient load-balancing techniques (*e.g.,* up to 4.03× MQPS) (§ IV-E).

II. Background and Motivation

In this section, we start with the background on key-value stores. Then we motivate general hotspot monitoring.

A. Background

Hotspot. Key-value store typically exhibits highly skewed workloads modeled using Zipfian distributions [24–26] with α>1 [3, 25]. Formally, in an epoch e, a hotspot h is a set of items $\{item_i\}_{i=1}^{n}$ whose count of accesses exceeds a given threshold T, $\sum_{i=1}^{n} count(item_i) > T$. For example, we consider breaking news suddenly occurs, which causes numerous accesses of the related items $\{item_i\}_{i=1}^{n}$. The threshold is defined as 1M. In a 1s epoch, the total count $\sum_{i=1}^{n} count(item_i) = 1.2M > 1M$, the set of items $\{item_i\}_{i=1}^{n}$ is regarded as a hotspot h. Hotspots are expensive, severe, and volatile so must be identified in real-time under strict latency SLOs. Firstly, hotspots may lead to business traffic drops and additional costs [4]. Secondly, hotspots could attract most of the accesses (*e.g.*, 90%) to only a few items (*e.g.*, 1%) [3]. Lastly, hotspots are volatile, *e.g.*, they might become unpopular in 10min [26]. Thus, hotspot monitoring is essential for hotspot mitigation.

Programmable Switches. The emerging programmable switches provide customizable line-rate packet processing (>10Tbps [23]) in the data plane (DP). These features allow us to customize hotspot identification on a switch and get the benefits of switching ASICs. The typical Reconfigurable Match-Action Tables (RMT) paradigm [27] maintains a pipeline of stages, each of which is equipped with an identical design, and 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. A P4 compiler maps the programs into the pipeline only when the required resources are sufficient.

Table I. Existing works identify workload types and overload locations

Overload Location (address)	WorkLoad Type	Solutions
Server Threads	Key	SwitchKV [6] Nap [15]
	Key/Op	ccNUMA [18] SPORE [21]
	Key/Op/Size	HotRing [3] MBal [8] MICA [20]
Cache	Key	Pacman [16]
Disk	Key	Anna [13]
	Key/Op/Size	HashKV [11]
Memory	Key	SlimCache [19]
	Key/Op	SwapKV [17]
	Key/Size	Segcache [12]
NIC	Key/Size	Minos [10]
Switch	Key	NetCache [5]
	Key/Op	Pegasus [7]

B. Special-Purpose Hotspot Monitoring

Recall that existing works customize hotspot monitoring for mitigations (section § I). As such, they focus on some specific hotspots and may overlook other ones.

Benchmark Analysis. To quantify the influence, we conduct a benchmark analysis. We build a key-value store system with three storage servers and a top-of-rack (ToR) switch. The switch is equipped with a read cache [5]. Each hotspot has corresponding workload types and overload locations. H1 is a write-intensive key that leads to the overloads in both L1 and process P1. H2 is a write-intensive key causing P1, P2, and server S2 overloads. H3 is a read-intensive key causing the overloads in the cache C1. We use consistent hash (CH) as the baseline. We assume this system is equipped with three mitigations targeting H1 and H2. M4 is not equipped.

- M1 (H1): scheduling requests to underutilized links.
- M2 (H2): re-routing requests to underutilized servers.
- M3 (H2): re-routing requests to underutilized processes.
- M4 (H3): filtering write-intensive items from the in-network cache to reduce read-only cache miss.

We inject three hotspots into this system by setting routing rules. We generate skewed workloads that follow a Zipf distribution with a skewness of 0.99. The workload distribution is the same as that of YCSB [28]. The content of items (a key-value pair from a search engine or an online shop) is unrelated to workload types and overload locations, both of which are mainly influenced by workload distributions. Thus, we generate the same distribution by YCSB as the work of real-world workloads like many existing works [3, 29–31]. Then, M1, M2, and M3 monitor and mitigate H1 and H2 respectively. As shown in Fig. 2, "+M2" means we deploy M1 and M2 on the system. Although the total throughput increases, H3 is the bottleneck that hinders the improvement of throughput. M1-M4 can improve throughput respectively. However, these monitoring approaches overlook H3 and fail to mitigate M4. If we generally monitor all hotspots rather than use a special-purpose monitoring approach, we have the opportunity to monitor H3 and further mitigate it.

C. General-Purpose Hotspot Monitoring

Generality Requirements. We investigate and identify the requirements in the existing works shown in Table I. (1) For *workload types*, existing works mainly identify content (*e.g.*, key), operations (*e.g.*, write or read), and value size (*e.g.*, long or short). They mitigate hotspots based on one or more workload types. For example, HotRing [3] mitigates head-of-line blocking for long-value items. Pegasus [7] replicates write- and read-intensive hotspots with different strategies. (2) For *overload locations*, a hotspot may lead to overloads across different locations in a system, such as server threads, Network Interface Cards (NICs), and a switch. They locate overloads for holistic mitigations of hotspots, *e.g.*, key replication, load migration [8] and content routing [6]. (3) Last, they need to identify the item keys of hotspots in all listed works. This requirement allows us to identify and track a hotspot precisely. Moreover, we also find evidence that the hotspot distributions have been observed in industrial-grade and large-scale systems at Meta [32] and Yahoo! [28]. This system monitors and finds the specific patterns of hotspots (*e.g.*, size and operator types) and overload locations (*e.g.*, caches, and servers).

Generality Definition. Motivated by these requirements, we define generality as querying the count value of a hotkey, workload types, overload locations, and any combination of these. Note that the definition of overload location can be at different levels. For example, users can define the overload locations at a cluster level (*e.g.*, a switch, and servers). They can also define the overload locations at a server level.

Example. We assume there are four workload types:

- (1) $load_w$: write type; (2) $load_r$: read type;
- (3) $load_s$: short-value type; (4) $load_l$: long-value type.

The overload locations include:

- (1) $addr_{phy}$: the physical address (*e.g.*, a server address);
- (2) $addr_{vir}$: the virtual address (*e.g.*, a process port);
- (3) $addr_{link}$: the link address (*e.g.*, a network port).

General hotspot monitoring should support query hotkeys and any combination of the above workload types and overload locations. We denote a query as $Q(f_1, f_2, ...)$, where f_1 and f_2 are fields such as $load_w$ and $addr_{vir}$, other fields remain unassigned. For example, one can query the count value of a hotkey $Q(key, ...)$, the count value of write-type keys in a server, $Q(key, load_w, addr_s, ...)$, the count value of all keys in a server by removing the key fields, $Q(addr_s, ...)$.

Remarks. The workload distribution is diverse in different use cases or applications. For example, keys are typically small, and large values only occur in special cases (*e.g.*, some databases) [32]. This makes general hotspot monitoring particularly important. (1) Without general hotspot monitoring, operators must make significant efforts to manually monitor and *confirm the presence of hotspots* in their use cases or applications. If this confirmation is not done, an operator may either waste substantial effort developing mitigation approaches for a non-existent hotspot or overlook an existing one, resulting in critical blind spots. (2) With general hotspot monitoring, there is no need for special-purpose hotspot monitoring in advance. Operators simply need to specify the types and locations of interest. They can then query any potential combinations and efficiently develop corresponding mitigation strategies. Furthermore, general hotspot monitoring removes blind spots by comprehensively covering all possible combinations, ensuring that no potential hotspots are missed.

III. SpotMon Overview

Our goal is to design a system for general hotspot monitoring. This system should support the implementation of programmable switches. Our design faces three challenges.

A. Challenges

Key Space (C1). General hotspot monitoring needs enormous key spaces. Because *any combination* of key, type, and locations could generate a large number of possible combinations. We consider items with 128-byte keys (8M possible keys) and 1024-byte values (1024 possible sizes) [20]. The item could be read-type, write-type, or not assigned (3 possible choices). It may go through 1 switch and 32 servers, each of which has 2 key-value store processes (1, 32, and 2 possible choices). To calculate the total number of combinations, we have $3 \times 1 \times 32 \times 2 \times 8 \times 10^6 \times 1024 = 1.572864 \times 10^{12}$ combinations. We need 5.72TB of memory to store all counter values with 4-byte counters in each monitoring epoch. The enormous key spaces consume non-trivial resource footprints.

Hotness Measurement (C2). Each key with types and locations indicates heterogeneous hotspots in a system. It is challenging to measure the hotness of different hotspots. For instance, a read-type hotspot could lead to an overload in a server. A write-type hotspot could lead to an overload in a switch. Both hotspots cause different kinds of overloads. Both have different types. Thus, it is not straightforward to compare which hotspot is hotter than another. We need an approach to measure the hotness of heterogeneous hotspots in a system.

Switch Constraints (C3). We face challenges when supporting general hotspot monitoring on a switch. First, the channel between a switch and a controller has limited bandwidth, which could be overwhelmed by a large amount of monitoring data [33]. Second, switch resources are very limited (*e.g.*, 12 pipeline stages with 10 MB TCAM and SRAM [27, 34, 35]). Third, the switch does not support complex computations, *e.g.*, loop, and multiple accesses of memory [27]. Thus, we need careful design to support general monitoring while coexisting with other switch functions (*e.g.,* firewalls and in-network key-value functionality, *e.g.,* cache).

B. Strawman Solutions and Lessons

Per-key Tracking (Alternative#1 (A1)). An alternative is to store all count values of combinations. However, it is expensive to store with server memory (C1) or with switch memory (*e.g.,* 10MB SRAM [27, 35], C3).

Sampling-based Monitoring (Alternative#2 (A2)). We can sample hotspot information and report periodically [6]. This reduces resource consumption but may lose significant information. In particular, under a low sample rate, it loses

Table II. The comparison of each alternative

Alternative	Accuracy	Resources
A1	High	**High**
A2	Low	**Median**
A3	High	**High**
SpotMon	**High**	**Low/Fast**

significant data (*e.g.,* the sudden change of workload distributions [6]). When increasing the sample rate, this approach gets close to A1. Furthermore, it may lead to drops under limited channel bandwidth (demonstrated in Exp#1). Last, operators must tune the sampling rate or aggregation intervals for the overhead-fidelity tradeoff, leading to heavy user burdens.

Sketch-based Monitoring (Alternative#3 (A3)). Sketch [36–38] is a data structure to reduce resource consumption. An alternative is to support each combination with a sketch [5]. However, each sketch may require between $O(KB)$ and $O(MB)$ of memory. A Tofino switch typically supports less than four sketches [22] or two sketches in our experiment (table V). It is far from the number of combinations.

Lessons. The combinations could consume non-trivial resource footprints. Simply storing all of these could easily consume a large number of system resources. We need a new abstract to reconsider this problem.

Architecture. SpotMon is a system for general hotspot monitoring in key-value stores. Fig. 3 shows the architecture of SpotMon. At a high level, SpotMon collects information about keys, types, and locations in storage servers. Then, the information is encoded to hotspot code with hierarchical encoding. The hotspot code is aggregated in a sketch within the switch data plane. At the end of each epoch, the control plane collects the sketch and decodes hotspots. Operators can use query interfaces to express their requirements and obtain the final hotspots. Note that the sketch can also be implemented in each storage server. We exhibit the design of SpotMon that is compatible with a programmable switch, which is equipped with fewer resources and has hardware constraints.

Step#1: Hotspot Encoding. We assign the number of bits for the hierarchical encoding without the need to know all keys in advance. For example, a storage server monitors item keys, workload types, and locations. Then, we take the general hotspot monitoring as an arbitrary partial spot query problem (C1 addressed). We encode the information into hotspot code. To enable these queries, SpotMon further provides a hierarchical hotspot encoding/decoding technique. It allows query hierarchically and reduces the overhead of hotspot search without exhausting queries. After encoding, the hotspot code is sent to a switch data plane.

Step#2: Hotspot Filtering. Hotspot code is aggregated in the switch data plane. Moreover, the switch deploys a sketch, which supports arbitrary partial spot queries (refer to section § IV). SpotMon filters hotspot candidates from hotspot codes. As such, only one sketch is needed to support general hotspot monitoring (C3 addressed). At the end of each epoch, the sketch is sent to the control plane.

Step#3: Hotspot Decoding. SpotMon decodes hotspots from the sketch. To measure the hotness of heterogeneous

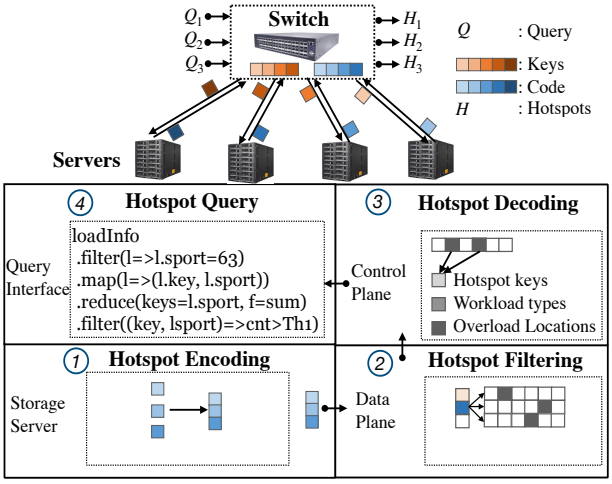

Fig. 3. **SpotMon Architecture**: SpotMon (1) hierarchically encodes hotspot candidates into hotspot code, (2) filters hotspot candidates from hotspot code, (3) decodes hotspots from candidates, and (4) queries hotspots with interfaces.

hotspots, we provide a new vector expression called hotspot skyline (C2 addressed). Hotspot skyline eliminates the one that is "colder" in all workload types and overload locations. Finally, SpotMon maintains a group of "hottest" hotspots.

Step#4: Hotspot Querying. To express the requirement of queries, we design a group of query interfaces for operators to freely express a wide range of monitoring tasks with familiar dataflow operators such as map, filter, and reduce. The query program will be parsed and executed in the control plane while benefiting from mitigations.

IV. Design

In this section, we elaborate on each module of architecture.

A. Hotspot Encoding

We first formulate general hotspot monitoring. Then, operators assign keys, types, and locations that may be identified in the future. This module encodes these fields to hotspot codes.

Problem Definition. We formulate general hotspot monitoring as an arbitrary partial key query problem [22]. We define it in the context of hotspot monitoring. Each field from keys, types, and locations can be taken as a partial spot. We can query any combination of partial spots. This definition meets our requirement of general monitoring.

Definition 1. *(Partial Spot). A spot s_p is a partial spot of full spot s_f (denoted by $s_p \prec s_f$), if there is a mapping $g(\cdot)$: $s_f \to s_p$, and for any element $e \in s_p$ defined on spot s_p, we have $f(e) = \sum_{e' \in s_f, g(e')=e} f(e')$, where $f(e)$ is a statistic of e.*

Example. Consider a $s_f = (key, load_w, addr_{phy})$. In the context of hotspot monitoring, a $s_f = (key_1, write, server_{31})$ means that a write-type key_1 is forwarded to $server_{31}$. Note that an s_p can be any subset of s_fs, *e.g.,* $(key, *, *)$ is an s_p of the s_f, where "$*$" is a wildcard.

Definition 2. *(Arbitrary Partial Spot Query). Given a s_f and a function f, return the $f(e)$ of any element $e \in s_p$ ($s_p \prec s_f$).*

Overloaded Locations				Item	Workload Types		
a_1	a_2	a_3	a_4	key	l_1	l_2	l_3

Fig. 4. **Hotspot Encoding**: The hierarchical encoding.

Example. Consider a $s_f = (key, load_w, addr_{phy})$ and count function f. It returns all s_p of the s_f, where a s_p is a combination in hotspot monitoring. This definition allows to freely query any s_p with s_fs. For example, we obtain the count of an s_p (*e.g.,* $(k_1, write, *)$) by aggregating all s_fs with the same loads (*e.g.,* $(key_1, write, *) = \sum_{i=0}^{31}(k_1, write, server_i)$). As such, we can query the count value of item keys $(k_i, *, *)$, types $(k_i, write, *)$, and overload locations $(k_i, *, server_i)$.

Insight. We observe that hotspots exhibit a feature of hierarchical aggregation. For instance, in fat-tree topology, there are two hotspots in both NICs in a server machine. Both hotspots in NICs also lead to an overload in this server. Moreover, if the hotspot is sufficiently popular, it could also lead to an overload in the top-of-rack (ToR) switch. To provide a unified and formal abstract of general monitoring, we encode keys, types, and locations following the hierarchical relationship. With such a top-down viewpoint, we can first query the fields with fewer combinations and reduce unnecessary queries. Note that it is unnecessary to encode the full location information of the whole data center and the identifiers of all devices. Rather, operators can choose their interested locations to monitor.

Encoding Fields. We exhibit typical fields, including keys, types, and locations. Each field is a binary code. The size of each field depends on the number of all possible values. For example, consider a field of servers $addr_{phy}$, which is encoded to a binary code. It is size is $log(|addr_{phy}|) = log(32) = 5$. Thus, we need 5 bits to encode this field.

Encoding Phase. We exhibit a typical encoding phase.

- **Fields (i): overload locations.** At a cluster level, keys can be aggregated from servers to switches. As such, we place the first location switch. Then, we place the second server. The insight behind this ordering is that a key forwarding to a server must go through a switch. Thus this key is likely to cause both locations overloaded.
- **Fields (ii): item keys.** Then, we place the field of item key after the fields of locations. Thus, we can query each key of hotspots after identifying the overload locations.
- **Fields (ii): workload types.** Finally, we append workload types behind the item key field. The ordering of workload types can follow the increasing order of field size. For example, we can place the read-write type before the size type. As such, we can query the short fields before the long ones to reduce unnecessary queries.

Location information, *e.g.*, IP addresses, or user-specific locations, *e.g.*, NICs, is encoded in hotspot code. Hotspot code is encapsulated in packet headers of responses. A switch can extract hotspot code from headers. Then, we encode hierarchical location information from a top-down viewpoint. Consider location information includes two locations, *i.e.*, storage machines and processes. We encode both locations in the order of cluster aggregation relationship, *i.e.*, machines and processes are encoded at the high- and the low-bit level.

Table III. Notations.

Symbol	Description
s_p	The partial spot of a hotspot candidate;
s_f	The full spot of a hotspot candidate;
$\mathcal{H}$	The hash table in the skyline part;
$h(.)$	The hash function of the hash table;
$\mathcal{H}[i]$	The i^{th} bucket of hash table;
$\mathcal{H}[i].K$	The key field in $\mathcal{H}[i]$;
$\mathcal{H}[i].V$	The value field in $\mathcal{H}[i]$;
$\mathcal{E}$	The exact table in the skyline part;
$\mathcal{E}[i]$	The i^{th} bucket of the exact table;
$\mathcal{E}[i].K$	The key field in $\mathcal{E}[i]$;
$\mathcal{E}[i].V$	The value field in $\mathcal{E}[i]$;
$\mathcal{C}$	The CocoSketch in the sky 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 sky part;
$\mathcal{C}_i[j]$	The j^{th} bucket in the i^{th} array in the sky part;
$\mathcal{C}_i[j].K$	The key field in $\mathcal{C}_i[j]$;
$\mathcal{C}_i[j].V$	The value field in $\mathcal{C}_i[j]$;
$P()$	Probability substitution function;

B. Hotspot Filtering

Next, we define the hotness measurement of a hotspot. Then, SpotMon filters and maintains hotspot candidates in a sketch. Then, the candidates are sent to the control plane for decoding.

Hotspot Skyline. We define hotspot skyline (HS) to measure the hotness of heterogeneous hotspots. Formally, HS is a vector v_k with s_p. A vector is denoted as $v_k = [f(s_{p1}), ..., f(s_{pn})]$, where $s_{p1}, ..., s_{pn}$ are the specific values of partial spots and $f(\cdot)$ is a count function.

Definition 3. *(v_1 dominates v_2). For two bounded vectors, $v_1 = [f(s_{p1}), ..., f(s_{pn})]$ and $v_2 = [f(y_{p1}), ..., (y_{pn})]$, $i \in [1, n]$, $f(s_{pi}) \geq f(y_{pi})$.*

Definition 4. *(Hotspot Skyline (HS)). HS is a set of v_k that cannot dominate each other.*

Remarks. Each s_p may be heterogeneous such that we cannot measure different s_p with a single value. Rather, we measure hotspots with multiple dimensions with a vector. When one hotspot dominates another, we can say the hotspot is "hotter" than the other one. For example, consider two vectors $v_1 = [f(read) = 3, f(server_1) = 5]$ and $v_2 = [f(write) = 4, f(switch_2) = 1]$. Because each value of a vector is not larger than that of another vector, both vectors are collectively referred to as HS. With this definition, we can measure the hotness of heterogeneous hotspots.

Lemma 1. *For a set M of any monotone scoring function h, $M \rightarrow R$, if a vector $v_k = [f(s_{p1}), ..., f(s_{pn})]$, where $v_k \in M$ maximizes $h(f(s_{p1}), ..., f(s_{pn}))$, then v_k is in the HS.*

Lemma 2. *For every vector $v_k = [f(s_{p1}), ..., f(s_{pn})] \in HS$, there exists a monotone scoring function $h(f(s_{p1}), ..., f(s_{pn}))$ such that v_k maximizes $h(f(s_{p1}), ..., f(s_{pn}))$.*

Lemma 2 indicates that every vector in *HS* expresses the hotness of s_p.

Remarks. Lemma 1 allows user to freely weigh hotness of each s_p and obtain an *HS*. Lemma 2 indicates that every

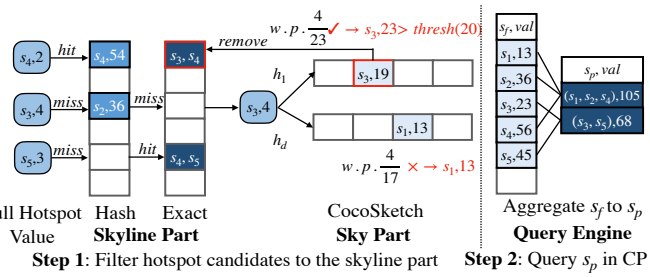

Fig. 5. **Hotspot Filtering**: (1) hotspot candidates are filtered to the skyline part, (2) hotspots can be queried in the control plane.

vector in *HS* expresses the hotness of s_p. The proof of both lemmas can be referred to [39–41]. Both lemmas mean HS is insensitive to the definition of HS. Users can freely weigh the s_p in their context.

Insights of Filtering. To support arbitrary partial spot queries, we leverage a sketch to maintain hotspot candidates. Specifically, our design is based on CocoSketch [22] and further customized for general monitoring. We filter hotspot candidates and maintain these candidates in the sketch. Hotspot filtering improves query accuracy and reduces resource consumption. Optionally, we can also choose other sketches sharing the same theory basis of subset-sum estimation [42] such as Unbiased SpaceSaving (USS) [43]. Note that we do not claim any novelty of the CocoSketch in our designs.

Sketch Data Structure. Fig. 5 shows that the sketch comprises a skyline part and a sky part. (1) In the *sky part*, we use a CocoSketch [22] to count s_fs while minimizing the stochastic variance of all s_fs. It maintains d arrays of l key-value pairs. Each array is associated with an independent hash function. It reports the s_f whose count exceeds a threshold as a hotspot candidate to the CP. Then, the candidates are inserted into the skyline part. (2) The *skyline part* contains a hash table $\mathcal{H}$ and an exact table $\mathcal{E}$. Each table entry records a s_f and its count value. With such two-layer filtering, the estimation errors are reduced. Moreover, the sky part also reduces hash collisions by removing some s_fs. Finally, the sketch is reported to the CP.

Hotspot Update Algorithm. For incoming data, SpotMon extracts s_f and the count of s_f. If the s_f was in the $\mathcal{H}$, SpotMon simply increments the counter of s_f (lines 1-2). Otherwise, SpotMon increments the counter of s_f if it was in the exact table (lines 3-4). If both tables were missed, SpotMon updated the sky part. SpotMon first maps the s_f to d buckets, each of which comes from one of the d arrays, and updates each bucket independently. Then SpotMon replaces the recorded s_f with a $\frac{s_f.cnt}{total\ value}$ probability. When the $total\ value$ exceeds a threshold, SpotMon removes the s_f exceeding the threshold and reports it to the CP. SpotMon inserts the s_f into the skyline part (lines 5-15).

Hotspot Query Algorithm. In the control plane, we build a table with two columns (s_f, $s_f.cnt$) by extracting the sketch. For the skyline part, we directly place the s_f and its count on the table. For the sky part, we take the median estimated size in different arrays as its final estimated count as the hardware version of CocoSketch [22]. The count of s_p is queried by aggregating all s_fs with the same s_p (Definition 2).

Algorithm 1 Update Algorithm

Input: s_f: full spot; $s_f.cnt$: the count of s_f; H; E; C;
Output: ▷ Update hotspot candidates in the skyline part
1: **if** s_f matches $\mathcal{H}[h(s_f)].\mathcal{K}$ **then**
2: $\quad\mathcal{H}[h(s_f)].\mathcal{V} = \mathcal{H}[h(s_f)].\mathcal{V} + s_f.cnt$
3: **else if** s_f matches $\mathcal{E}[h(s_f)].\mathcal{K}$ **then**
4: $\quad\mathcal{E}[h(s_f)].\mathcal{V} = \mathcal{E}[h(s_f)].\mathcal{V} + s_f.cnt$
5: **else** ▷ Update statistics in the sky part
6: $\quad$**for** each row i in the sky part **do**
7: $\quad\quad val\ idx = h_i(s_f)$
8: $\quad\quad\mathcal{C}_i[idx].\mathcal{V} = \mathcal{C}_i[idx].\mathcal{V} + s_f.cnt$
9: $\quad\quad val\ pr = \frac{s_f.cnt}{\mathcal{C}_i[idx].\mathcal{V}}$
10: $\quad\quad\mathcal{C}_i[idx].\mathcal{K} = P(\mathcal{C}_i[idx]).\mathcal{K},\ s_f,\ pr)$
11: $\quad\quad$**if** $\mathcal{C}_i[idx].\mathcal{V} > threshold$ **then** ▷ Filter hotspots
12: $\quad\quad\quad$insert $\mathcal{C}_i[idx]$ to the skyline part
13: $\quad\quad$**end if**
14: $\quad$**end for**
15: **end if**

Example. Fig. 5 shows how SpotMon inserts a s_f into the sketch. For a s_f in both $\mathcal{H}$ and $\mathcal{E}$, such as s_6 and s_5, SpotMon increments its counter. For other s_fs, both s_1 and s_3 are counted in the sky part. SpotMon increments the counter and replaces the s_f with a probability $\frac{s_f.cnt}{total\ value}$. At the end, s_fs are reported to the CP. When querying s_p (s_3, s_5), SpotMon aggregates both $ls3$ and s_5 to produce the final result 64.

Stochastic Variance Minimization. We first analyze the variance sum of loads for minimization of load estimation sum.

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

By separating the skyline and other s_fs, SpotMon has high fidelity. The sky part is based on the theory of stochastic variance minimization. We denote the ground truth 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).

In the sky part, we consider the incremental minimization of the sum of variance caused by a load of each insertion. We ensure that the variance change caused by each insertion is minimized. Thus SpotMon reduces the overall errors of s_p.

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

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

Proof. SpotMon filters some s_fs from the sky part. Each insertion in the skyline part does not introduce additional hash collisions. So the the variance sum does not increase. Therefore, SpotMon minimizes the increase of variance sum. □

Error Bound. Next, we discuss the estimation error of SpotMon. Consider a CocoSketch of $d * l$ in the sky 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 sky part.

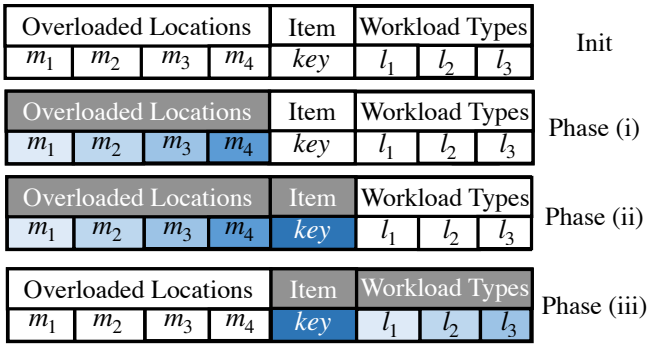

Fig. 6. **Hotspot Decoding**: the hierarchical decoding phases.

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

$$\mathbb{P}\left[R(e) \geqslant \epsilon \cdot \sqrt{\frac{f(\bar{e})}{f(e)}}\right] \leqslant \delta \tag{3}$$

Proof. For hotspot candidates stored in the skyline part, errors exist only due to hash collisions in the sky part before insertion into the skyline part. Since the sky part is dominated by cold loads, there are fewer hash collisions and fewer errors are generated. At the same time, the value in the sky part accounts for a small proportion of the hot loads, so the error of the hot loads is negligible. In the worst case, a bucket in the skyline part is not updated after insertion. The error bound of $R(e)$ can still be obtained by Lemma 1. □

C. Hotspot Decoding

Next, hotspot candidates are decoded into hotspots for further querying with a top-down algorithm.

Insights. With hierarchical encoding, SpotMon leverages the aggregation feature to decode hotspot codes with a top-down algorithm. SpotMon decodes hotspot codes to eliminate unnecessary decoding operations. For example, it can first check whether a server is overloaded or not. It can stop decoding later codes if this server is not overloaded. As such, we can quickly decode hotspots and improve query efficiency.

Decoding Phases. As shown in Fig. 6, SpotMon decodes hotspots from coarse to fine granularities. To decode the s_f, we first identify overloaded locations. Then, for each overloaded location, we identify the keys that make key contributions to the overload. Further, we identify the workload types of these keys to support precise hotspot mitigations. This process stops when no result returns in the current phase. As such, we reduce unnecessary operations and decode hotspot results.

- **Phase (i): general decoding.** We first decode overload locations, keys of items, and workload types to a s_f. More precisely, we can decode the location according to the aggregation relation of the topology. For example, the items first accumulate in a switch, Then, these items transfer to more network cards and processes. As such, we can first decode switch locations (*e.g.*, ports, cache entry IDs). Then, we decode the IDs of network cards and process IDs. Once a location is overloaded, we can stop and decode the hotspot in this location.

Table IV. SpotMon Query Primitives

Primitives	Descriptions
`loadInfo`	Information of monitoring data.
`filter(R, pred)`	Output in `R` under predicate `pred`.
`map(R, [exprs], [fields])`	Map expressions, `[exprs]`, into fields of R, and output tuples with new fields, `[fields]`.
`reduce(R, func`	Aggregate the value of `R` with a function `func`.

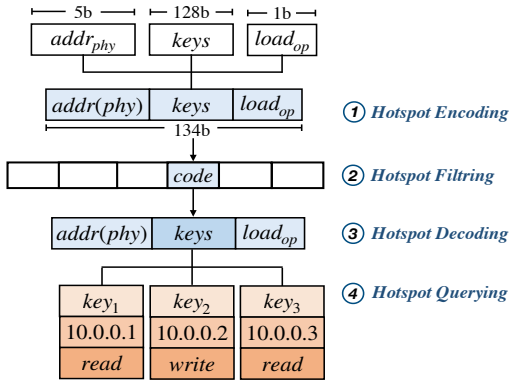

Fig. 7. **Case Study**: hotspot encoding, filtering, decoding, and querying.

- **Phase (ii): overload locations.** To decode a s_f, we first identify the overload locations such as switch ports. If the count of s_p exceeds a threshold, the s_p is treated as an overloaded location. A system operator can estimate the maximum count of items in a module and define the threshold. For example, as shown in Fig. 6, SpotMon queries each system location from m_1 to m_4, returning the addresses of locations. If no address is returned, we stop all phases and return no output.
- **Phase (iii): hotspot identification.** If there is an overload location, we will identify hotspots in this location. Specifically, We assemble the key and locations as an s_p. As such, we can decode the hotspots in an overloaded location by checking the counts that exceed a threshold.
- **Phase (iv): workload types.** Next, we decode each hotspot along with their workload types. We assemble both workload types and keys as an s_p. The count of s_p exceeds a threshold and is identified as the hotspot with workload types. As such, operators can tailor specific mitigations according to their workload types.

D. Hotspot Querying

With the decoded hotspots, operators can express their query requirements for the hotspots of interest with interfaces.

Skyline Query. After decoding all hotspots, SpotMon queries hotspot skylines from these hotspots. Specifically, it checks every hotspot and finds the hotspot that can be dominated by a hotspot. The dominated hotspots will be removed from the set of hotspots. Finally, we obtain the set of hotspot skylines. Operators can mitigate the "hottest" spots with corresponding mitigation techniques.

Query Interfaces. SpotMon provides a unified abstract of the query. System operators express the query requirements with a simple interface. The description of the interface is

Table V. (Exp#2/#4) Hardware resource utilization of SpotMon, which count 5 s_p and reduces resource consumption than A3 (multiple sketches). SK1 measures hotkeys and SK2 measures hotkeys and hot processes using NetCache [5]. We show the resource usage of SpotMon (SP). We list the resource reduction with SK1 (R1) and SK2 (R2).

Resource Type	SK1(%)	SK2(%)	SP(%)	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

shown in Table IV. We show an example of a query program and omit the detailed description due to the space limit.

```
loadInfo
.filter(l=>field=(f1, f2))
.map(l=>(f1, f2))
.reduce(keys=(f1, f2), f=sum)
.filter(l.(f1, f2)=>cnt>Th.(f1, f2))
```

E. Case Study

In this section, Fig. 7 showcases how SpotMon performs general hotspot monitoring, including four steps.

- *Hotspot Encoding.* Operators assign partial spots $addr_{phy}$, $keys$, and $load_{op}$. We assume there are 32 servers (5b). The size of the item key is 128b. The operation type $load_{op}$ has two values (1b), read type and write type. Thus, we assign 134b for each hotspot code.
- *Hotspot Filtering.* Each hotspot code (134b) is compared with hotspot candidates in the sketch. The sketch filters hotspot candidates from other codes.
- *Hotspot Decoding.* Hotspot codes are decoded to s_f, with the algorithm of hierarchical decoding.
- *Hotspot Querying.* Operators can query hotspots from these decoded hotspots with interfaces. For example, they query three hotspots in the case.
- *Hotspot Mitigation.* By identifying hotspots generally, SpotMon can identify write-intensive items and allows M4 to reduce cache misses and improve system performance (up to 4.03×MQPS).

V. Implementation and Hardware Optimizations

Next, we provide implementation optimization of SpotMon.

Skyline Part Aggregation. We split s_fs into slices and compare whether the s_p is identical to the one stored in the hash table. However, we can sacrifice a little control-plane bandwidth by aggregating the $\mathcal{H}$ into the $\mathcal{E}$. Specifically, for each hotspot candidate, we report to the CP and insert it into the $\mathcal{E}$. The consumption of control-plane bandwidth is negligible because we report to the CP only when an s_p is first monitored as a heavy load. Further, we can leverage the longest prefix match (LPM) to accelerate the HS match. If multiple s_p exceeded thresholds, we can set the corresponding bits in a bitmap to one. Then, we further leverage LPM to match specific rules while benefiting from TCAMs.

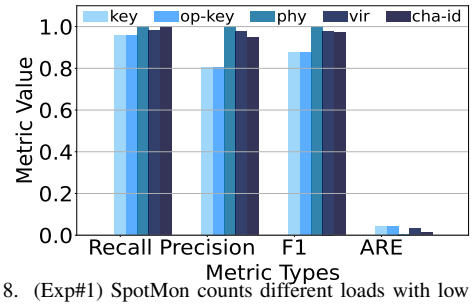

Fig. 8. (Exp#1) SpotMon counts different loads with low errors.

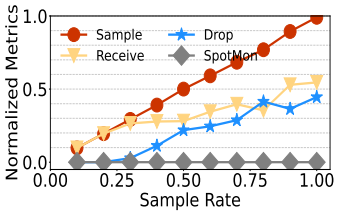

Fig. 9. (Exp#2) SpotMon is resource-efficiency compared to A2 (Drop), which causes more packet drops with higher sampling rates.

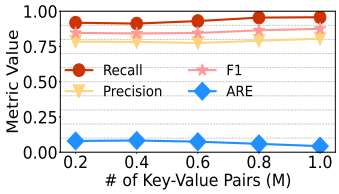

Fig. 10. (Exp#5) SpotMon maintains low estimation errors under the different number of requests.

Metadata Dependence. The length of s_f may exceed the maximum bit width of a register array. To store s_f, we split a s_f into slices and use multiple arrays to store each slice, respectively. However, the multiple register arrays may depend on one metadata to index the location of a s_f in the array. As a result, a register array must wait for another one to process the metadata because multiple register array depends on one metadata. The operations on these arrays cannot be in parallel, consuming a large number of pipeline stages.

Parallelize Index Calculation. To remove the dependence, we *parallelize* the calculation of the index based on the RMT architecture. 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 for the total resource budgets. We demonstrate these in our evaluation (Exp#1).

Parameter Configuration Users may have different resource budgets and SLOs, they can configure SpotMon with SketchGuide [35] to maximize SLOs given resource budgets.

VI. Evaluation

We maintain the same definition of hotspots in § II-A. We evaluate the monitoring accuracy (Exp#1), resource efficiency (Exp#2), hardware resource footprint (Exp#3), and scalability across different processing points, requests, and machines (Exp#4–6), and robustness under varying workload skewness and write ratios (Exp#7–8).

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.

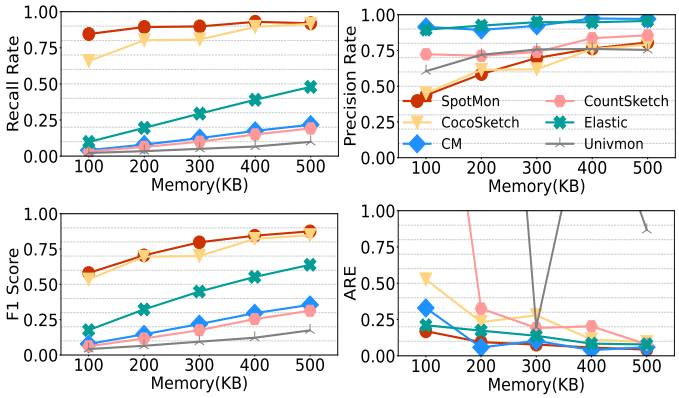

Fig. 11. (Exp#2) SpotMon has low errors under different memory budgets.

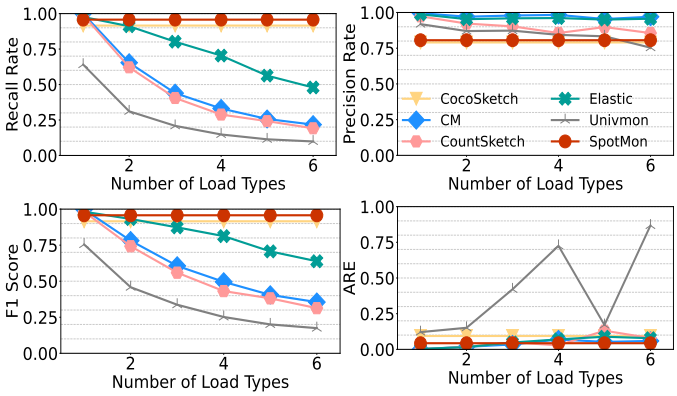

Fig. 12. (Exp#4) SpotMon has low errors under the different # of s_p.

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 [6, 44] and are evidenced by real-world deployments [28]. We use Zipf 0.99 in most experiments to demonstrate that SpotMon provides high accuracy even under extreme scenarios. The workload distribution is the same as that of YCSB [28]. We also show that SpotMon still provides low estimation errors under less-skewed workloads (*e.g.,* Zipf 0.8). We generate queries under a Zipf distribution in our clients [6]. We use 16-byte keys and 128-byte values. SpotMon can be deployed on a switch or a server machine. It supports 128-byte keys on a server machine but may be inefficient to be deployed on a switch. Because 128 bytes could consume a large amount of PHV. Thus, we keep the same setup as NetCache. The keyspace is hash partitioned across storage servers [45]. The write ratio ranges from 0.1 to 0.9. To evaluate general monitoring, we monitor the case in section § II.

Metrics. The normalized metric indicates the percentile of the total packets. We identify the keys whose count values exceed a given threshold as the ground truth. We denote TP as true positive, FN as false negative, 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) $\sum_{e \in \Psi} \frac{|f(e)-\hat{f}(e)|}{f(e)}$. We measure hardware resources SRAM, SALU, Hash calls, and Stages.

Baselines. We implement two alternatives A2 and A3. A1 is the special case of A2 (when the sampling rate is 1). The typical work of A2 includes NetCache [5]. The typical works of A3 include Pegasus [7] and SwitchKV [6]. Moreover, we compare the monitoring errors with CocoSketch [22], CountMin sketch [38], Elastic sketch [36], Univmon [37], and CountSketch [46]. We run SpotMon 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 [22]. We set the threshold to be 10^{-4} of the total counts of loads.

Setup. We maintain the same encoding approach in § IV-E. We implement SpotMon in a Tofino switch [23] with 500KB memory. The skyline part is optimized using the techniques in § IV.F. The exact table has 1280 entries. The residual memory budget is all consumed by the CocoSketch in the sky part.

Exp#1: General Monitoring. Fig. 9 shows we showcase that SpotMon supports general hotspot monitoring. We count five s_p with SpotMon (§ II), including item keys ($keys$), the combination between operators and keys ($operator, key$), the address of server ($addr_{phy}$), the address of process ($addr_{vir}$), and the address of channel ($addr_{cha}$). Overall, SpotMon keeps high accuracy for the queries on s_p. Because SpotMon filters hotspot candidates in the skyline part it improves query accuracy. We observe that a s_p with large key spaces may consume more memory footprints and impose more hash collisions. Thus, it may have more estimation errors.

Exp#2: Resource Efficiency. We compare with A2 in § III-B, where A1 is a special case of A2 when the sampling rate is one. We send 1M packets by Pktgen [47] with a 10K rate. The data plane samples packets and reports to the switch control plane. Fig. 10 shows that as the sample rate increases, the number of dropped packets increases. Because A2 needs to sample a great number of packets to the control plane. The limited bandwidth causes the packet drops (up to 50%). SpotMon maintains a low drop rate because only a sketch with limited memory consumption will be sent to the control plane. This consumes very limited bandwidth. As shown in Fig. 12, we change the memory budgets and keep other setups the same. SpotMon is more accurate than other sketches in the same memory budgets. Moreover, SpotMon has lower ARE than CocoSketch because the skyline part filters the heavy loads from the sky part two to ensure the accuracy of the heavy loads. The ARE results of some sketches have some oscillations due to the specific workload features.

Exp#3: Hardware Resources. Table V shows that SpotMon is resource-efficiency. We take the monitoring part of NetCache (*i.e.*, the CM, and the Bloom filter) as the baseline and eliminate other functions. We follow the same configuration in the original paper. For each spot, we increment the number of CM. Table V shows that (1) the hotkey (SK1) and (SK2) the hotkey and the $addr_{phy}$ (SK2). SK2 consumes expensive 91.67% of stages and fails to compile 3 loads (key_w). SpotMon counts all 5 spots while consuming a proportion of hardware resources. Moreover, SpotMon reduces the resource consumption significantly except SRAM and Meter ALU

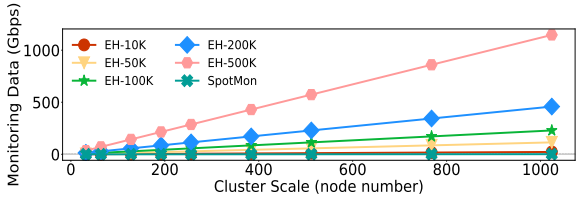

Fig. 13. (Exp#6) SpotMon is scalable to # of endhosts (EHs) and # of requests.

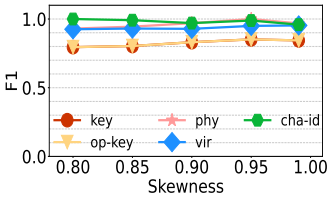

Fig. 14. (Exp#7) SpotMon has low errors under different load skewness.

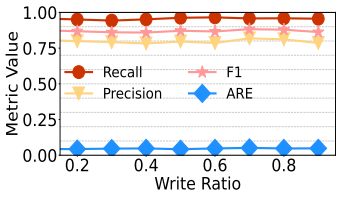

Fig. 15. (Exp#8) SpotMon maintains low error under different write ratios.

than SK1. But SpotMon does not consume more resources than SK2 while monitoring 3 more spots. SpotMon achieves resource efficiency due to stochastic variance minimization, which reduces the errors of all s_fs.

Exp#4: Partial Spot Scalability. Fig. 13 shows that we compare with A3 in § II, we gradually add new s_p for each sketch. Fig. 13 shows that as the number of s_p 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 s_p (poor recall rate). SpotMon and CocoSketch maintain high scalability in all the accuracy metrics. SpotMon always has higher performance than CocoSketch because SpotMon filters hotspots from the sky part. The ARE of UnivMon has a little oscillation due to the specific workload distribution.

Exp#5: Request Scalability. Fig. 11 shows that we change the number of requests to test scalability. As the number of requests increases, the estimation errors remain stable or slightly reduced. SpotMon is scalable because the skyline part prevents s_fs from sharing the same counter thereby accurately counting the hotspot. Moreover, the skyline part also increases the accuracy when the number of key-value pairs increases because the heavy loads could be filtered to the skyline part.

Exp#6: Cluster Scalability. Fig. 14 shows that SpotMon is highly scalable for large clusters. We increase the number of hosts in a cluster from 32 to 1024. The monitoring data is expressed in a 128-bit key and a 32-bit value. Each host reports to the controller from 10K to 500K pairs each time for every 100ms. The small number of pairs indicates that fewer pairs are sampled. At scale, both alternatives report a significant amount of data, overwhelming the controller channel. While SpotMon reports negligible data, which has been aggregated in the switch. SpotMon is scalable because only one SpotMon is required for each 32-node rack and reported to the CP.

Exp#7: Workload Skewness. Fig. 15 shows that we count 5 different s_ps to test the robustness. 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 skyline part. Thus, SpotMon is robust when the workload distribution is highly skewed.

Exp#8: Write Ratio. Fig. 16 shows we count each metric when the write ratio ranges from 0.2 to 0.9. Each metric remains stable, exhibiting high robustness in read- and write-intensive workloads. When querying the write-intensive workloads, SpotMon aggregates the partial loads with the same operations. The identification of operations is only 1 bit but SpotMon keeps high accuracy due to stochastic variance minimization, which customizes for subset sum estimation.

VII. Discussion

SpotMon does not assume deployment locations. It is hardware-friendly and can be deployed on both a server and a switch. There are three benefits to deploying SpotMon on a switch. (1) It is compatible with switch-based works [5, 7]. (2) The switch enables line-rate monitoring (*i.e.,* >6.5Tbps [23] of packet processing rates). (3) This capability makes monitoring more scalable for more overload locations (§ VI Exp#6).

VIII. Related Works

A. Hotspot Monitoring

Existing works monitor hotspots at servers [6, 8, 9, 11–21], channels [10], and switches [5, 7], to enable mitigations. However, they customize monitoring and cause blind spots (see § II). SpotMon enables efficient general hotspot monitoring without customizing monitoring approaches.

B. Heavy-Hitter Detection

Some works such as ColdFilter [48] detect single-dimension heavy hitters [48–50]. Some works detect heavy hitters in multiple dimensions [22, 51–56]. The work [55] introduced a simple algorithm for multiple-dimension heavy hitters. R-HHH [51] detects heavy hitters based on the type of prefixes given an application. CocoSketch [22] provides a sketch for multiple-dimension heavy hitters. However, it remains unexplored to apply these works to general hotspot monitoring in key-value stores. SpotMon provides efficient and general hotspot queries with encoding and decoding.

IX. Conclusion

We present SpotMon, a system that supports efficient general hotspot monitoring in key-value stores. With SpotMon, operators can identify hotspots with workload types and overload locations. SpotMon allows users to comprehensively monitor hotspots and design holistic mitigations. Under different workload distributions, SpotMon is scalable and robust due to the hotspot encoding, filtering, and decoding. Moreover, SpotMon consumes acceptable resources with high accuracy. In the future, we plan to scale SpotMon to multiple racks and enable more mitigation techniques.

X. Acknowledgement

We thank our shepherd Prof. Patrick P. C. Lee and the anonymous reviewers for enlightening comments. Chunming Wu and Dong Zhang are the corresponding authors. This work is supported by the "Pioneer" and "Leading Goose" R&D Program of Zhejiang (2024C01066). This work is supported by the Research Project of Provincial Laboratory of Shandong, China under Grant No. SYS202201. This work is supported by Quan Cheng Laboratory (Grant No. QCLZD202304).

REFERENCES

[1] (2017) Redis lab. redis. [Online]. Available: https://https://redis.io/N

[2] (2012) Matthew shafer. memcached. [Online]. Available: https://github.com/memcached/memcached

[3] 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.

[4] (2006) Greg linden. akamai online retail report. [Online]. Available: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. of SIGCOMM'18*, 2018.

[10] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores." in *Proc. of NSDI'19*, 2019.

[11] H. H. Chan, Y. Li, P. P. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in kv storage via hashing," in *Proc. of ATC'18*.

[12] J. Yang, Y. Yue, and R. Vinayak, "Segcache: a memory-efficient and scalable in-memory key-value cache for small objects." in *Proc. of NSDI'21*, 2021.

[13] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," *Proc. of VLDB*, 2019.

[14] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores." in *Proc. of ATC'19*, 2019.

[15] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to numa-aware persistent memory indexes." in *Proc. of OSDI'21*, 2021.

[16] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu, "Pacman: An efficient compaction approach for log-structured key-value store on persistent memory," in *Proc. of ATC'22*.

[17] L. Cui, K. He, Y. Li, P. Li, J. Zhang, G. Wang, and X. Liu, "Swapkv: A hotness aware in-memory key-value store for hybrid memory systems," *IEEE TKDE*, 2021.

[18] V. Gavrielatos and e. Katsarakis, "Scale-out ccnuma: Exploiting skew with strongly consistent caching," in *Proc. of EuroSys'18*.

[19] Y. Jia, Z. Shao, and F. Chen, "Slimcache: An efficient data compression scheme for flash-based key-value caching," *ACM Transactions on Storage (TOS)*, 2020.

[20] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *Proc. of NSDI'14*, 2014.

[21] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. of SoCC'13*, 2013.

[22] 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.

[23] (2022) Barefoot tofino 2. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/

[24] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. of SIGMETRICS*, 2012.

[25] 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.

[26] B. Berg and e. Berger, "The $cachelib$ caching engine: Design and experiences at scale," in *Proc. of OSDI'20*, 2020.

[27] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, and e. McKeown, Nick, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *SIGCOMM CCR*, 2013.

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

[29] Z. Zhu, Y. Zhao, and Z. Liu, "In-memorykey-value store live migration with netmigrate," in *Proc. of FAST'24*.

[30] Y. Xu, H. Zhu, P. Pandey, A. Conway, R. Johnson, A. Ganesan, and R. Alagappan, "Ionia: High-performance replication for modern disk-based kv stores," in *Proc. of FAST'24*.

[31] Q. Zhang, Y. Li, P. P. Lee, Y. Xu, and S. Wu, "Depart: Replica decoupling for distributed key-value storage," in *Proc. of FAST'22*, 2022, pp. 397–412.

[32] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *Proc. of FAST'20)*, 2020.

[33] J. Rasley and e. Stephens, "Planck: Millisecond-scale monitoring and control for commodity networks," *SIGCOMM CCR'14*.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. of ICDE*, 2001.

[40] H.-T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," *Journal of the ACM (JACM)*, 1975.

[41] F. P. Preparata and M. I. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[42] N. Duffield, C. Lund, and M. Thorup, "Priority sampling for estimation of arbitrary subset sums," *Journal of the ACM (JACM)*, vol. 54, no. 6, pp. 32–es, 2007.

[43] 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.

[44] 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.

[45] D. Karger and e. Lehman, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. of SOTC*, 1997.

[46] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[47] Pktgen. [Online]. Available: https://github.com/danieltt/pktgen

[48] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate

stream processing," in *Proc. of SIGMOD'18*, 2018.
[49] H. Han, Z. Yan, X. Jing, and W. Pedrycz, "Applications of sketches in network traffic measurement: A survey," *Information Fusion*, vol. 82, pp. 58–85, 2022.
[50] Z. Zhou, D. Zhang, and X. Hong, "Rl-sketch: Scaling reinforcement learning for adaptive and automate anomaly detection in network data streams," in *Proc. of LCN'19*, 2019.
[51] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proc. of SIGCOMM*, 2017.
[52] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 155–166.
[53] ——, "Finding hierarchical heavy hitters in streaming data," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 4, pp. 1–48, 2008.
[54] J. Hershberger, N. Shrivastava, S. Suri, and C. D. Tóth, "Space complexity of hierarchical heavy hitters in multi-dimensional data streams," in *Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2005, pp. 338–347.
[55] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," in *Proc. of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2012, pp. 160–174.
[56] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proc. of SIGCOMM Conference on Internet Measurement*, 2004.