Hare: A Systematic Framework for Efficient and Generally Automatic Hotspot Offloading on Programmable Switches

Xueying Zhu, Yingtao Li, Xiang Li, Jialin Li, Zeke Wang

***Abstract*—Switch-based hotspot offloading is a trendy solution for latency-sensitive applications to achieve high system throughput with an acceptable P99 query response latency. However, due to the varying object sizes, dynamic workloads, and complex query-processing functions of the latency-sensitive applications, existing switch-based dynamic hotspot offloading approaches struggle to handle these applications effectively. This is mainly because of their inefficient switch resource utilization and non-generalizable hotspot offloading designs. So we propose Hare, a systematic framework that consists of three techniques to address these issues. First, Hare uses a MAT-based cross-stage structure to store and perform hit-checks for large hotspots on the switch data plane. Second, Hare uses a switch-server co-offloading mechanism to support fast and precise offloading. Third, Hare is designed to enable generally automatic offloading by decoupling application-related query processing with hotspot offloading. Compared to the state-of-the-art approaches, Hare supports $8.86\times \sim 9.97\times$ larger hotspot size, achieves $1.27\times \sim 6.61\times$ higher system throughput, and can recover the system throughput and the P99 query response latency within 8s.**

***Index Terms*—query processing, programmable switch, tail latency SLO, in-network caching**

I. Introduction

Services that cater to clients, such as web search engines, financial trading platforms, games, and online social networks, necessitate consistently low response times to attract and retain users [1]–[3]. A primary design objective for client-facing services is to maximize query throughput while meeting tail latency SLOs for individual queries [4]–[8]. The importance of low tail latency comes from the observation that query tail latency greatly impacts the client's experience and the company's business revenues. Take Amazon online web services for example: tail latency SLO violations delay web page loading times, and each additional 100 milliseconds of query tail latency causes a $1.5 million decrease in revenue [9], [10]. Moreover, a large online service that usually contains multiple micro-service components [11]–[13] requires stricter tail latency for each component like a key-value store (KVS) system, whose tail latency should be hundreds of microseconds to a few milliseconds [14]–[16].

A recent line of work [17]–[20] increases throughput and reduces tail latency by caching frequently accessed objects on high-speed programmable switches. Although these works show promising results, we still identify three challenges when applying in-network caching to real-world workloads:

C1. Varying Object Sizes. Object size distribution across different workloads exhibits significant variance. For instance, the lock ID length in a lock manager system is as short as 4 bytes [18], while the key length in a key-value store system can range from tens to hundreds of bytes [21]–[23]. However, existing solutions place rigid constraints on the offloaded object sizes, limiting their generality: the maximum switch object size in NetCache [17] is limited by the match action table (MAT) bit width in a single pipeline stage (typically 128 bytes); DySO [20] relaxes this restriction by storing large objects in registers across stages, but the bit width per stage is still bounded by the register array size (typically 16 bytes).

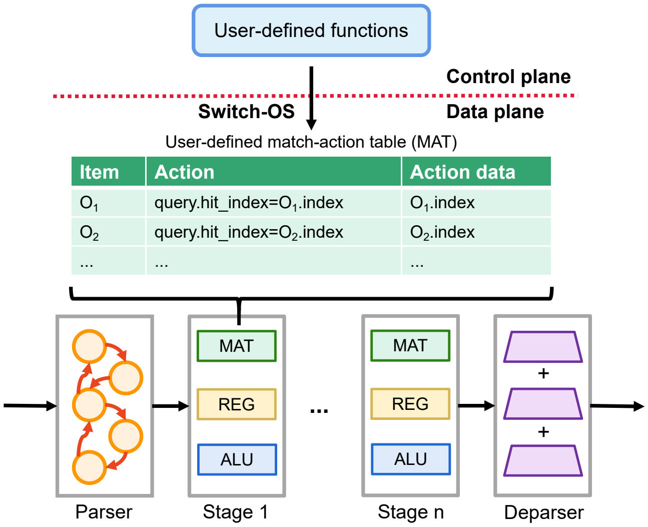
C2. Dynamic Workloads. Object popularity changes rapidly due to unpredictable real-world incidents [24]–[29]. To maintain low tail latency, an in-network caching solution should quickly react to popularity changes. However, offloading an object in NetCache takes milliseconds on average. This large overhead arises from inefficient switch-OS-based APIs used to realize interactions between the switch control plane and the switch data plane (hereinafter called plane interactions). Besides, when workload changes, NetCache's recovered switch hit rate is limited by the precision of the space-saving statistical modules on the switch data plane. DySO uses a register-based structure called *r-MAT* to replace the MAT structure in order to support fast microsecond-level offloading with packet-based plane interactions. However, DySO's recovered switch hit rate is limited by the hash collision when offloading objects to the r-MAT structure.

C3. Complex Query-processing Functions. To achieve high throughput, applications usually require the switch data plane to maintain dirty data[1] for offloaded objects and don't synchronize them with backend servers [18], [19]. It makes realizing hotspot offloading sticky because the offloading mechanism should keep data consistent while replacing cold offloaded objects with hot non-offloaded objects. However, NetCache is designed as a read-intensive KVS system, while DySO is designed as a general hotspot offloading mechanism for read-only systems. Both of them don't allow dirty data on the switch data plane, so their offloading process can't be directly adopted

X. Zhu, Y. Li, X. Li, Z. Wang are with the Department of Computer Science, Zhejiang University, China. Email: {zhuxueying, Li_Yingtao, lixiang3, wangzeke}@zju.edu.cn.

J. Li is with the National University of Singapore, Singapore. Email: lijl@comp.nus.edu.sg.

[1] We adopt the concept of "dirty data" from caching. In the scenario of hotspot offloading, it refers to data that is solely stored on the switch without being written back to the backend servers.



Item	Action	Action data
O_1	query.hit_index=O_1.index	O_1.index
O_2	query.hit_index=O_2.index	O_2.index
...	...	...

Fig. 1. Protocol Independent Switch Architecture (PISA)

by write-intensive applications (e.g., write-intensive KVS, lock manager, and OLTP).

To address issues in existing approaches, we present Hare, a systematic framework for efficient and automatic hotspot offloading on programmable switches. It consists of the following three key innovations:

K1. MAT-based Cross-stage Hotspot Storage Structure. To enable high flexibility in hotspot size, we present e-MAT, a hotspot storage structure utilizing MATs in multiple stages to store and perform hit-checks for large offloaded objects.

K2. Switch-server Co-offloading Mechanism. To realize fast and precise offloading, we present a switch-server co-offloading mechanism. For precision, it gathers statistics from both the switch and the backend servers to find the hottest non-offloaded objects and the coldest offloaded objects. For fastness, it uses real-time min-heap, hybrid plane interactions, and batch offloading to reduce the offloading delay.

K3. Automatic Offloading Mechanism. To achieve high programmability, we present an automatic offloading mechanism, which decouples application-related query-processing from the hotspot offloading mechanism, such that developers focus on application-related functions while leaving Hare to automatically manage the dynamic hotspot offloading process.

We implement a read-only KVS and a lock manager with Hare. Compared to state-of-the-art approaches, Hare supports $8.86\times \sim 9.97\times$ larger hotspot size, achieves $1.27\times \sim 6.61\times$ higher system throughput, and can recover the system throughput and the P99 query response latency within 8s.

II. Background

Programmable switches are an emerging trend in the market with specialized ASICs from vendors such as Intel [30] and Cavium [31]. Compared to traditional switches [32], [33], programmable switches provide the capabilities of flexible packet processing at a line rate of up to billion pkt/s. The programmability of the switch data plane is enabled by a configurable architecture [34] and the de facto data plane programming language is P4 [35]. P4 is a high-level language that allows users to write match-action rules to express the processing of packets on the data plane. P4 uses a C-like syntax with several restrictions. For example, it does not allow the use of pointers, loops, and complex mathematical functions which impedes P4 programs running at a line rate.

To run P4 programs, programmable switches often use the PISA architecture, as shown in Figure 1, to allow programmers to specify how a packet should be processed using MATs, registers, and ALUs on the switch data plane. Packet processing happens concurrently in a sequence of stages, each of which handles a packet at a time. The resources in each stage are limited and a packet usually cannot access again the stage it has already passed. As a result, the layout of functions on the data plane must be carefully organized so that dependent operations are arranged in successive order. The switch device provides an embedded CPU board that serves as the control plane for the switch. During runtime, the switch control plane can modify MATs and registers using Switch-OS-based APIs, while packets can only modify registers on the data plane.

The PISA architecture contributes to in-network caching typically in the following manner: it employs MATs to store offloaded objects and to perform hit-checks, and uses registers to store the associated application data. When a query is considered a hit, it is processed directly within the switch data plane using the associated application data. We take NetCache as an example to show how the PISA architecture works. In this case, the offloaded objects are keys and the application data are values. As shown in Figure 1, NetCache stores hot keys (e.g., O_1, O_2, ...)[2] in the MAT structure as items, and programs match-action rules for each item in the MAT. To be specific, if the queried key in a query packet matches any item (e.g., O_1) of the MAT, the corresponding action for this item is to assign the action data (e.g., $O_1.index$) to a query-related variable (e.g., $query.hit\_index$) which will be further used to locate the application data in registers.

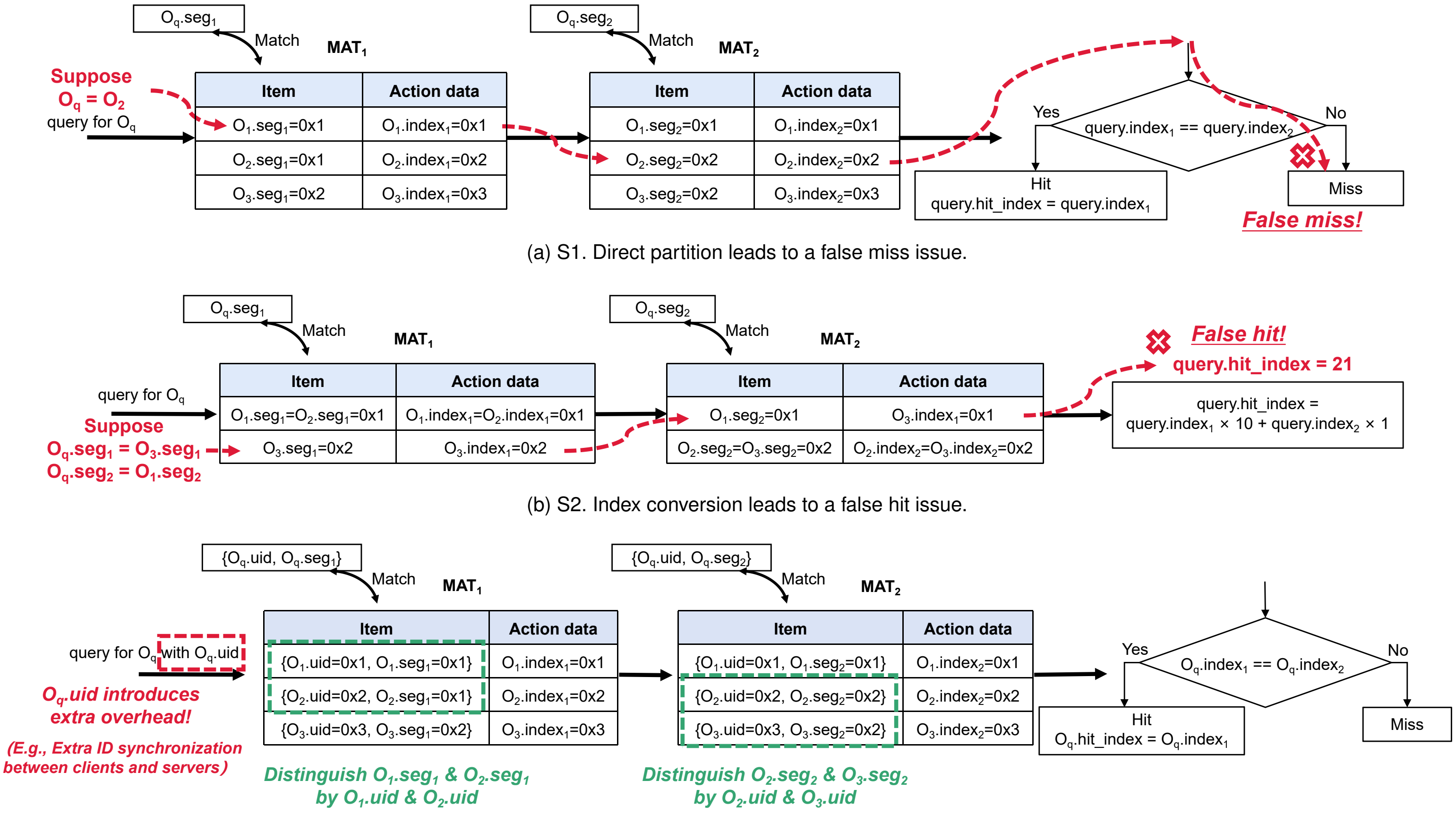
III. Motivation

We identify three issues that prevent existing works from wide adoption by a broad range of applications.

A. Low Flexibility in Hotspot Size

The distribution of object sizes varies significantly across different workloads. While applications such as the lock manager typically handle small objects (e.g., 4-byte lock IDs), there are still specific scenarios where handling large objects (e.g., hundred-byte keys) is common. We take two application scenarios that usually require large objects as examples: In distributed file systems, the full file path can be used as a key for querying the file metadata [36], [37]. In scenarios with complex directory hierarchies, this can lead to large key sizes. Similarly, in Internet of Things (IoT) databases, dataset identifiers are employed as keys for querying dataset locations. These identifiers typically include detailed information such as device owner, device name, device location, measurement type, time range, and so on, resulting in large key sizes [38]. Offloading query processing functions of these applications

[2] O is abbreviated for "object".



O_q.seg_1
Match
MAT_1
Suppose
O_q = O_2
query for O_q
Item
Action data
O_1.seg_1=0x1
O_1.index_1=0x1
O_2.seg_1=0x1
O_2.index_1=0x2
O_3.seg_1=0x2
O_3.index_1=0x3
O_q.seg_2
MAT_2
O_1.seg_2=0x1
O_1.index_2=0x1
O_2.seg_2=0x2
O_2.index_2=0x2
O_3.seg_2=0x2
O_3.index_2=0x3
Yes
No
query.index_1 == query.index_2
Hit
query.hit_index = query.index_1
Miss
False miss!
query for O_q
Suppose
O_q.seg_1 = O_3.seg_1
O_q.seg_2 = O_1.seg_2
O_1.seg_1=O_2.seg_1=0x1
O_1.index_1=O_2.index_1=0x1
O_3.seg_1=0x2
O_3.index_1=0x2
O_1.seg_2=0x1
O_3.index_1=0x1
O_2.seg_2=O_3.seg_2=0x2
O_2.index_2=O_3.index_2=0x2
False hit!
query.hit_index = 21
query.hit_index = query.index_1 × 10 + query.index_2 × 1
{O_q.uid, O_q.seg_1}
{O_q.uid, O_q.seg_2}
query for O_q with O_q.uid
O_q.uid introduces extra overhead!
(E.g., Extra ID synchronization between clients and servers)
{O_1.uid=0x1, O_1.seg_1=0x1}
O_1.index_1=0x1
{O_2.uid=0x2, O_2.seg_1=0x1}
O_2.index_1=0x2
{O_3.uid=0x3, O_3.seg_1=0x2}
O_3.index_1=0x3
Distinguish O_1.seg_1 & O_2.seg_1 by O_1.uid & O_2.uid
{O_1.uid=0x1, O_1.seg_2=0x1}
O_1.index_2=0x1
{O_2.uid=0x2, O_2.seg_2=0x2}
O_2.index_2=0x2
{O_3.uid=0x3, O_3.seg_2=0x2}
O_3.index_2=0x3
Distinguish O_2.seg_2 & O_3.seg_2 by O_2.uid & O_3.uid
O_q.index_1 == O_q.index_2
O_q.hit_index = O_q.index_1

(a) S1. Direct partition leads to a false miss issue.

(b) S2. Index conversion leads to a false hit issue.

(c) S3. Additional item field leads to extra overhead.

Fig. 2. Naive solutions to extend the maximum size of the offloaded keys with MATs do not work.

to the switch data plane requires support for large objects. However, existing solutions place rigid constraints on the offloaded object sizes, limiting their generality.

1) NetCache: NetCache uses the MAT structure on the switch data plane to store and perform hit-checks for offloaded objects. When the queried object hits any offloaded object in the MAT structure, NetCache processes the query directly on the switch data plane. However, NetCache's hit-check mechanism restricts the maximum size of an offload object to the maximum bit width of the MAT structure in only one pipe stage of the switch data plane. To illustrate the challenges in extending object size, we present the following three naive solutions that leverage MATs across multiple stages to support larger offloaded object sizes.

Naive Solution S1: Direct Partition. As shown in Figure 2a, we can directly partition a large offloaded object (e.g., O_1) into several segments (e.g., $O_1.seg_1$ and $O_1.seg_2$) and use MATs in different stages to store different segments of the object as items (e.g., MAT_1 stores $O_1.seg_1$, MAT_2 stores $O_1.seg_2$). The items that are stored in different MATs but belong to the same object share the same action data (e.g., $O_1.index_1 = O_1.index_2 = 0x1$). When a query arrives, each MAT checks if the corresponding segment of the queried object O_q matches any item. If items matched by O_q in MATs share the same action data, the query is determined to be a hit and will be handled by the switch data plane. However, when a MAT has an item that belongs to at least two objects, this mechanism would cause a false miss issue because of the matching priority for these items. Figure 2a illustrates an example of sequentially offloading segments of O_1, O_2, and O_3 to two MATs. We assume that the incoming object O_q is O_2 and that $O_2.seq_1$ is equal to $O_1.seq_1$. Because the first offloaded item has the highest priority for matching, O_q will match $O_1.seg_1$ in MAT_1 and $O_2.seg_2$ in MAT_2. As a result, the query will be finally determined to be a miss.

Naive Solution S2: Index Conversion. To address the issue of S1, we employ the index conversion approach to avoid duplicate items in a MAT, as shown in Figure 2b. The corresponding index of the offloaded segment that O_q matches in each MAT will be multiplied by a unique bias (e.g., $\times 10$, $\times 1$) and then summed. The summed result is used as the final index ($query.hit\_index$) to locate the application data in registers. However, this approach would lead to a false hit issue when the item matched by O_q in each MAT doesn't belong to the same offloaded object. Figure 2b illustrates an example where $O_q.seg_1 = O_3.seg_1$ and $O_q.seg_2 = O_1.seg_2$. In this example, O_q gets a valid final index and causes a false hit issue.

Naive Solution S3: Additional Item Field. Another way to address the issue of S1 is to introduce an additional item field for each segment in each MAT. As shown in Figure 2c, each offloaded object has a globally unique ID (e.g., $O_1.uid \neq O_2.uid \neq O_3.uid$), and each segment in each MAT takes the corresponding object's unique ID as an additional item field. In this way, if two or more objects have the same segment in a MAT (e.g., $O_1.seg_1 = O_2.seg_1 = 0x1$ in MAT_1), the unique ID can be used to distinguish which segment belongs to which object, so as to avoid the false miss issue caused by duplicate items. However, though this

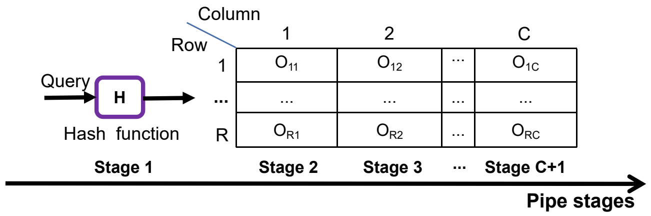


Fig. 3. The r-MAT structure. (O_{rc} is the offloaded object in the r-th row and the c-th column.)

solution does solve the false miss issue, it requires the client to include the unique ID of the queried object in the query packet, resulting in extra overhead. For example, the client should either send extra packets to retrieve the unique ID of the queried object from servers, or waste memory to store the unique ID locally. Since maintaining a lightweight client is important for user experience, especially on mobile devices with limited computing, networking, and memory resources [39], [40], this solution is not optimal.

2) DySO: DySO proposes a register-based structure called r-MAT to store and perform hit-checks for offloaded objects, with the assumption that the size of the offloaded object fits in the bit-width of the register array in one stage so each stage can provide one column. Figure 3 illustrates that the registers in multiple stages build a two-dimensional matrix in an r-MAT structure. When a query arrives, the stage for hashing decides which row of the matrix will interact with the query. Then the queried object will be compared with the corresponding offloaded object in each stage. If the queried object matches any offloaded object, the query will be determined to be a hit.

Although DySO doesn't explicitly explain how to offload objects whose size exceeds the bit-width of the register array in a single stage, we can easily implement it by using multiple stages to represent one column. In particular, one offloaded object can be partitioned into several segments and each segment occupies a stage with the same row index. However, DySO still struggles to expand the size of the offloaded objects because the r-MAT structure consumes stages heavily. Typically, the maximum bit-width of the registers in one stage is $\frac{1}{8}$ of the maximum bit-width of the MAT in one stage. As a result, compared to the MAT structure, the r-MAT requires 8× stages to store a large offloaded object. Given that modern switches usually have only 10 $\sim$ 20 stages [18], the stage overhead makes r-MATs impractical for large offloaded objects.

B. Not Supporting Fast and Precise Offloading

To cope with dynamic workloads, the controller needs to frequently update the switch data plane with hotspots. The primary challenge is the limited switch table and register update rate. While commodity switches can update more than 10K table entries per second [41], the update rate is insufficient to support traditional cache update mechanisms like LRU and LFU. These mechanisms update the cache for every query, causing unnecessary in-network cache churns and performance degradation [17], [42]. To avoid unnecessary cache churns, both NetCache and DySO offload an object to the switch data plane only when it becomes hot enough, rather than for each query access. However, it is challenging to accurately identify the top-K hotspots and quickly offload them to the switch data plane. In the following content, we explain why NetCache and DySO can't achieve fast and precise offloading.

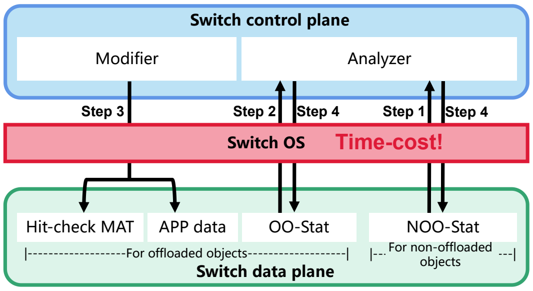


Fig. 4. NetCache's dynamic hotspot offloading process.

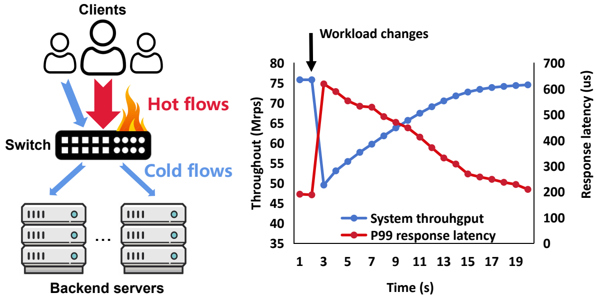


Fig. 5. The impact of slow offloading on the system throughput and the query response latency.

1) NetCache: NetCache has a high offloading delay because the switch-OS-based plane interactions dominate the overall time cost of the NetCache's offloading process. Figure 4 shows NetCache's offloading process can be separated into four steps, each of which involves costly plane interactions. In Step 1, the NOO-Stat[3] reports one hot object to the analyzer. In Step 2, the analyzer identifies a cold object by scanning the OO-Stat[4]. In Step 3, the modifier modifies the object in the hit-check MAT and the application data in registers. NetCache repeats Step 1 $\sim$ Step 3 multiple times before proceeding to Step 4. In Step 4, the analyzer clears the OO-Stat and the NOO-Stat. In particular, NetCache spends milliseconds to offload one hotspot, and plane interactions take up more than 90% of the total time.

Figure 5 shows NetCache's slow hotspot offloading causes a wide peak in the P99 query response latency and a wide valley in the system throughput with 32k offloaded objects under the Zipf-0.99 distribution. Besides, NetCache's NOO-Stat needs to balance between the expected precision and the required resources and thus limits the stable switch hit rate.

2) DySO: DySO realizes fast offloading at the sacrifice of precision because of the r-MAT hash collision. Figure 6 shows DySO's offloading process. In Step 1, the analyzer fetches recorded queried objects from the QO-Record[5]. In Step 2, the analyzer updates the local statistics with the recorded objects,

[3] NOO-Stat (Statistical Module for Non-offloaded Objects) is the statistical module to store and update frequency counts for non-offloaded objects. NetCache implements a Count-Min sketch [43] to report hot non-offloaded objects, and a Bloom filter [44] to remove duplicate reports.

[4] OO-Stat (Statistical Module for Offloaded Objects) is the statistical module to store and update frequency counts for offloaded objects. NetCache implements a precise counter for each offloaded object.

[5] QO-Record (Module for Recording Queried Objects) is a module that records query objects without distinguishing between offloaded and non-offloaded objects.

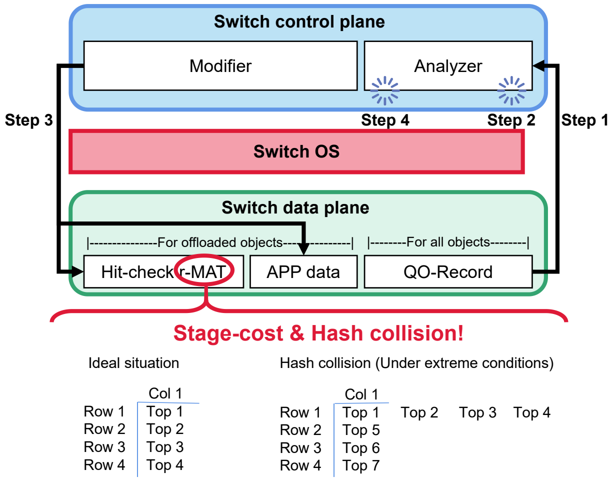


Fig. 6. DySO's dynamic hotspot offloading process.

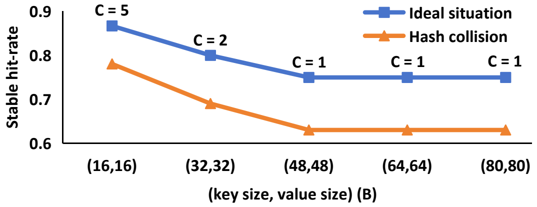


Fig. 7. The stable hit rate of the DySO-based read-only KVS system. (C is the number of columns of the r-MAT structure.)

and checks if the set of the hottest objects changes with the updated statistics. In Step 3, the modifier modifies the hit-check r-MAT and the application data to ensure the hottest objects and the related application data are offloaded. In Step 4, after a certain period of time (e.g., 1s), the analyzer ages the frequency counts of all objects. DySO loses precision on hotspot offloading because it only supports offloading the local hottest objects for each row of the r-MAT structure instead of offloading the global hottest objects due to the hash collision.

Figure 6 explains how hash collision impacts the precision of the hotspot offloading. For a 4×1 r-MAT, the ideal situation is to offload the top $1 \sim$ top 4 hottest objects. However, assuming the top $1 \sim$ top 4 hottest objects share the same hash index, only the top 1 hottest object can be offloaded. Since the number of rows of the r-MAT is highly limited to a small number (e.g., 32k), the hash collision is severe. DySO eases the impact of the hash collision by increasing the number of columns. However, because the r-MAT structure is stage-cost, it is difficult to expand the number of columns when dealing with large offloaded objects and application data. We implement a DySO-based read-only KVS system and evaluate its stable hit rate under Zipf-99 query distribution with various key and value sizes, as shown in Figure 7. We observe that when increasing the key size and the value size, the switch data plane supports an obviously fewer number of columns, and the hash collision increases the gap between the ideal hit rate and the real hit rate.

C. Customized Design

To achieve high throughput, applications usually require the switch data plane to maintain dirty data for offloaded objects and don't synchronize them with backend servers in real time [18], [19]. It makes realizing hotspot offloading sticky because the offloading mechanism should keep data consistent while replacing cold offloaded objects with hot non-offloaded objects. However, existing works fail to ensure data consistency when dirty data exists during hotspot offloading, as we discuss in the following content.

1) NetCache: NetCache is customized for a read-intensive KVS system, which typically has two types of queries: GET and PUT. The GET query reads the value of the queried key, and the PUT query modifies the value of the queried key. NetCache uses the switch data plane as a read-only cache and employs the write-through policy for PUT queries, which means the PUT query always invalidates the offloaded matched key on the switch and will be transferred to the backend servers. Because the system is read-intensive, the write-through policy hardly affects the system throughput. Also because of the write-through policy, the backend servers in NetCache always have the latest version of all key-value pairs, so deleting the cold offloaded key and its value on the switch data plane won't cause data inconsistency. However, for other systems (e.g., lock manager [18], OLTP system [19]) whose application data (e.g., the phase of the lock) is write-intensive, they can't tolerate write-through policy for each query that participates in application data modification. As a result, they require application data mitigation from the switch data plane to the backend servers before deleting the cold offloaded object and its related application data. As such, NetCache's offloading process no longer functions properly.

2) DySO: DySO is a general mechanism for read-only systems (e.g., the read-only KVS system and the network address translation system). As NetCache, DySO also directly deletes the cold offloaded object and its related application data on the switch data plane during hotspot offloading, since read-only systems do not have dirty data on the switch data plane. However, for write-intensive applications that have dirty data on the switch data plane, this offloading process can't be applied, as it would result in the loss of dirty data.

IV. Design of Hare

A. Overview of Hare

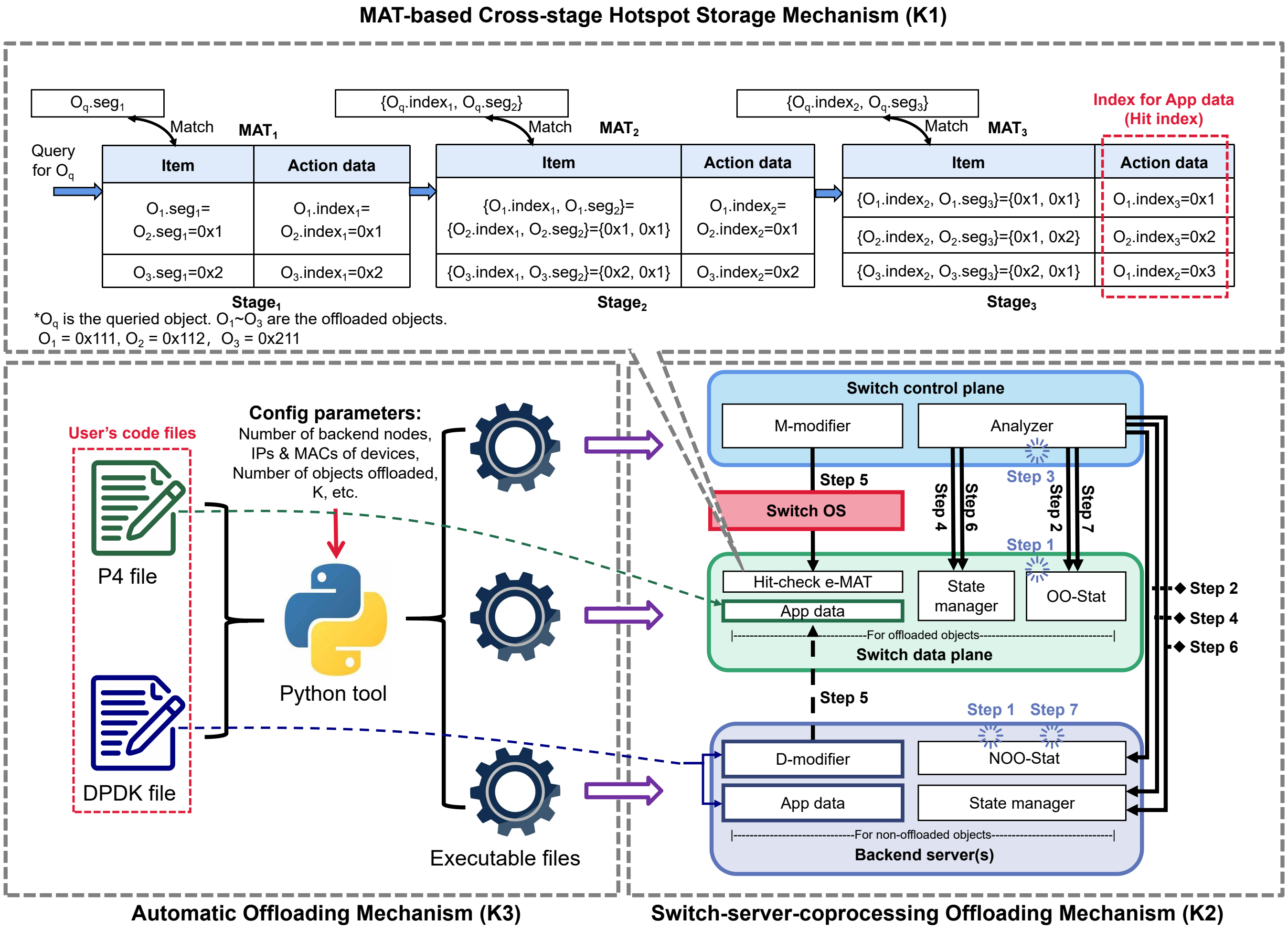
According to the drawbacks of existing works, when designing Hare, we keep the following goals in mind:

G1. High Flexibility in Hotspot Size. Because the number of stages on the switch data plane is highly limited, Hare should allow offloading large objects in a stage-saving way.

G2. Fast and Precise Offloading. Because the available space for storing offloaded objects on the switch data plane is highly limited, Hare should precisely identify and offload the hottest objects. Meanwhile, Hare should also offload hotspots fast to the switch data plane to avoid wide peaks and valleys in the P99 query response latency and the system throughput.

G3. Generally Automatic Offloading. To allow offloading hot objects across diverse applications, Hare should be designed as a framework that ensures generally automatic offloading while maintaining data consistency for various use cases.

System overview. To achieve the above goals, we propose Hare, a systematic framework for efficient and automatic



Item	Action data
$O_1.seg_1$= $O_2.seg_1$=0x1	$O_1.index_1$= $O_2.index_1$=0x1
$O_3.seg_1$=0x2	$O_3.index_1$=0x2

Item	Action data
$\{O_1.index_1, O_1.seg_2\}$= $\{O_2.index_1, O_2.seg_2\}$={0x1, 0x1}	$O_1.index_2$= $O_2.index_2$=0x1
$\{O_3.index_1, O_3.seg_2\}$={0x2, 0x1}	$O_3.index_2$=0x2

Item	Action data
$\{O_1.index_2, O_1.seg_3\}$={0x1, 0x1}	$O_1.index_3$=0x1
$\{O_2.index_2, O_2.seg_3\}$={0x1, 0x2}	$O_2.index_3$=0x2
$\{O_3.index_2, O_3.seg_3\}$={0x2, 0x1}	$O_1.index_2$=0x3

*O_q is the queried object. O_1~O_3 are the offloaded objects.
O_1 = 0x111, O_2 = 0x112, O_3 = 0x211

Fig. 8. Design overview of Hare. Hare consists of three key ideas K1, K2, and K3.

hotspot offloading on programmable switches. Figure 8 illustrates the overview of Hare, which has three key innovations:
K1. MAT-based Cross-stage Hotspot Storage Structure. To achieve high flexibility in hotspot size (G1), we propose e-MAT, a new hotspot storage structure using MATs in multiple stages to store and perform hit-checks for offloaded objects.
K2. Switch-server Co-offloading Mechanism. To realize fast and precise offloading (G2), we design a new offloading mechanism. For precision, it gathers statistics from both the switch data plane and the backend servers to identify the hottest non-offloaded objects and the coldest offloaded objects. For fastness, it uses real-time min-heap, hybrid plane interactions, and batching optimization to reduce offloading delays.
K3. Automatically Offloading Mechanism. We present a new development process to realize generally automatic offloading (G3). It decouples application-related query-processing with hotspot offloading so developers are only required to fulfill application-related code regions as developing a system without tedious dynamic hotspot offloading.

B. MAT-based Cross-stage Hotspot Storage Structure

For latency-intensive applications that can benefit from hotspot offloading, the object size distribution across different workloads exhibits significant variance. However, the existing solutions place rigid constraints on the offloaded object sizes as mentioned in Subsection III-A. So we propose e-MAT, a new hotspot storage structure that aims to extend the scalability of the MAT structure with functional correctness, where the functional correctness requires (1) no false miss – no duplicate items in each MAT structure, and (2) no false hit – no unexpected hit for non-offloaded objects.

Key Insight for E-MAT. The key insight behind constructing the e-MAT structure is to introduce an appropriate additional item field for segments in MATs. The field should identify which offloaded object a segment belongs to while not imposing extra overhead on the client, as discussed in the naive solution S3 in Subsection III-A. Intuitively, the index from the previous MAT suits this job well for the following reasons: First, the index of each segment in a MAT can be set to a unique value within the MAT. This allows the subsequent MAT to identify which offloaded object a certain segment belongs to, by using the index from the previous MAT as the additional item field. Figure 9 illustrates how it works. Suppose O_1 $(0x111)$ and O_2 $(0x212)$ are two offloaded objects, and both of them can be divided into three equal-length segments. $O_1.index_1$ from MAT_1 is used as an additional item field for O_1's corresponding item $\{O_1.index_1, O_1.seg_2\}$ in MAT_2. Similarly, $O_1.index_2$ from MAT_2 is used as an additional item field for O_1's corresponding item $\{O_1.index_2, O_1.seg_3\}$ in MAT_3. The same logic applies to O_2. When a query arrives, each MAT checks if the corresponding segment of the queried object with the previous index matches any item. If a query

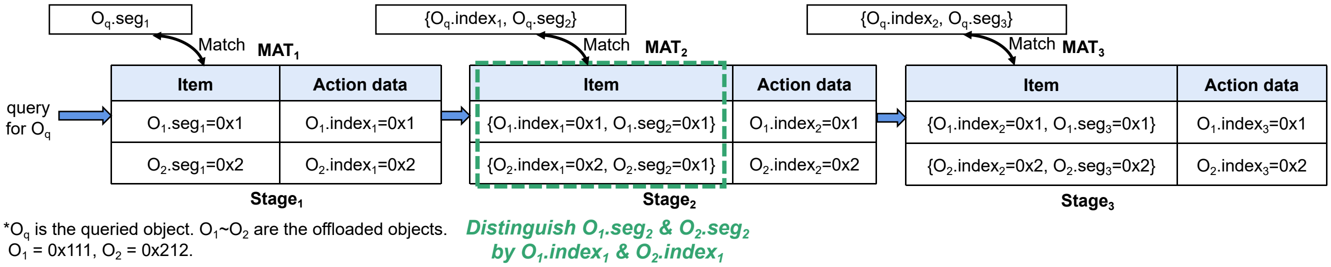

$O_q.seg_1$
Match
MAT_1
query for O_q
Item
Action data
$O_1.seg_1$=0x1
$O_1.index_1$=0x1
$O_2.seg_1$=0x2
$O_2.index_1$=0x2
Stage$_1$
{$O_q.index_1$, $O_q.seg_2$}
Match
MAT_2
{$O_1.index_1$=0x1, $O_1.seg_2$=0x1}
$O_1.index_2$=0x1
{$O_2.index_1$=0x2, $O_2.seg_2$=0x1}
$O_2.index_2$=0x2
Stage$_2$
{$O_q.index_2$, $O_q.seg_3$}
Match
MAT_3
{$O_1.index_2$=0x1, $O_1.seg_3$=0x1}
$O_1.index_3$=0x1
{$O_2.index_2$=0x2, $O_2.seg_3$=0x2}
$O_2.index_3$=0x2
Stage$_3$
*O_q is the queried object. O_1~O_2 are the offloaded objects.
O_1 = 0x111, O_2 = 0x212.
Distinguish $O_1.seg_2$ & $O_2.seg_2$ by $O_1.index_1$ & $O_2.index_1$

Fig. 9. Key insight for E-MAT: Use the index from the previous MAT as the additional item field.

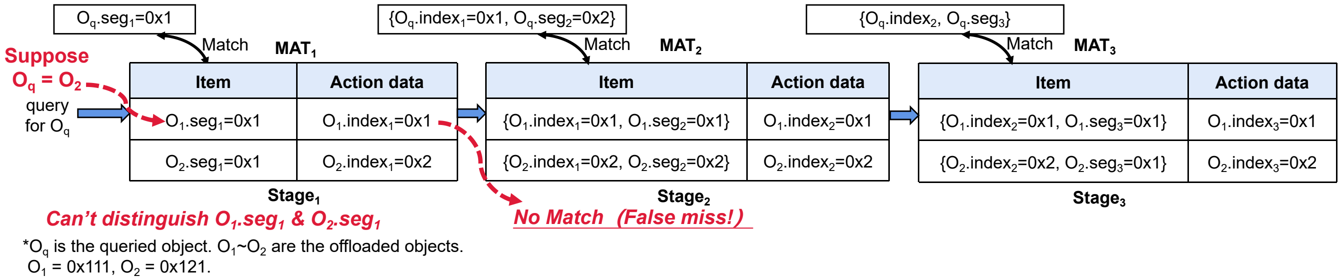

$O_q.seg_1$=0x1
Match
MAT_1
Suppose O_q = O_2
query for O_q
Item
Action data
$O_1.seg_1$=0x1
$O_1.index_1$=0x1
$O_2.seg_1$=0x1
$O_2.index_1$=0x2
Stage$_1$
Can't distinguish $O_1.seg_1$ & $O_2.seg_1$
{$O_q.index_1$=0x1, $O_q.seg_2$=0x2}
Match
MAT_2
{$O_1.index_1$=0x1, $O_1.seg_2$=0x1}
$O_1.index_2$=0x1
{$O_2.index_1$=0x2, $O_2.seg_2$=0x2}
$O_2.index_2$=0x2
Stage$_2$
No Match (False miss!)
{$O_q.index_2$, $O_q.seg_3$}
Match
MAT_3
{$O_1.index_2$=0x1, $O_1.seg_3$=0x1}
$O_1.index_3$=0x1
{$O_2.index_2$=0x2, $O_2.seg_3$=0x1}
$O_2.index_3$=0x2
Stage$_3$
*O_q is the queried object. O_1~O_2 are the offloaded objects.
O_1 = 0x111, O_2 = 0x121.

Fig. 10. Directly introducing the index from the previous MAT as an additional item field leads to a false miss issue.

has a matching item in each MAT, it is determined to be a hit query and the index from the last MAT will be used to index the application data. It's worth noting that though O_1 and O_2 have the same second segment ($O_1.seg_2 = O_2.seg_2 = 0x1$), there is no false miss issue caused by duplicate items in MAT_2 because previous indexes $O_1.index_1$ and $O_2.index_1$ help distinguish $O_1.seg_2$ and $O_2.seg_2$ in MAT_2. There is also no false hit because the additional item field links the segments of the same offloaded object one by one across different MATs. As a result, a query can find a matching item in each MAT only if it queries for an offloaded object. Second, in the PISA structure of the programmable switch, a query can easily collect indexes while passing through MATs in a pipeline manner on the switch data plane, without imposing additional overhead on the client.

Though the key insight behind the e-MAT structure is straightforward, it's still challenging to solve the false miss issue caused by duplicate items in MAT_1, because no index from the previous MAT can be used to distinguish segments in MAT_1. For example, as shown in Figure 10, suppose O_1 equals $0x111$, O_2 equals $0x121$, and O_1's matching priority is higher than O_2. We can see that when the queried object O_q equals O_2, $O_q.seg_1$ $(0x1)$ will match $O_1.seg_1$ $(0x1)$ in MAT_1, while $\{O_q.index_1, O_q.seg_2\}$ $(\{0x1, 0x2\})$ won't match any item in MAT_2, causing a false miss issue. It's because no index from the previous MAT can be used to distinguish $O_1.seg_1$ and $O_2.seg_1$ in MAT_1.

E-MAT Construction Mechanism. Distinguishing duplicate items in MAT_1 causes extra overhead on the client as mentioned in Subsection III-A. However, is it the only way to avoid the false miss issue? The answer is no. To avoid the false miss issue caused by duplicate items, Hare builds the e-MAT structure by merging duplicate items in each MAT. For example, as shown in Figure 8, in MAT_1 of the e-MAT structure, we merge $O_1.seg_1$ and $O_2.seg_1$ because both of them equal $0x1$. It's worth noting that merging duplicate items (e.g., $O_1.seg_1$ and $O_2.seg_1$) in a MAT will further trigger the merging of their corresponding indexes (e.g., $O_1.index_1$ and $O_2.index_1$). Similarly, in MAT_2, we merge $\{O_1.index_1, O_1.seg_2\}$ and $\{O_2.index_1, O_2.seg_2\}$ because both of them equal $\{0x1, 0x1\}$. It's important to merge duplicate items in each MAT, not just in MAT_1, because merged items and their corresponding merged indexes may cause duplicate items in the following MAT. By merging duplicate items in each MAT, we can solve the false miss issue because there are no duplicate items in each MAT. There is also no false hit issue because O_q can pass through all hit-checks only when O_q is identical to an offloaded object.

E-MAT Updating Mechanism. It's not easy to fast and safely update the e-MAT structure. We use the following two naive solutions to show the difficulty: First, naively replacing the offloaded object's segments in the e-MAT structure with the non-offloaded object's segments is fast, but not safe. For example, in Figure 8, if we replace O_1 $(0x111)$ with O_4 $(0x411)$ by directly replacing $O_1.seg_1$ $(0x1)$ with $O_4.seg_1$ $(0x4)$, we can find that O_2 $(0x112)$ is also replaced by an irrelevant non-offloaded object $(0x412)$. It's because $O_1.seg_1$ and $O_2.seg_1$ merge, and they share the same item in MAT_1. Second, naively reconstructing the e-MAT structure is safe but not fast. Specifically, reconstructing the e-MAT structure includes the following steps: (1) Analyze the new set of items that should exist in the e-MAT. (2) Delete all items in the e-MAT. (3) Insert the new set of items to the e-MAT. Define M as the number of MATs in the e-MAT structure and N as the number of offloaded objects. This process involves $M \times N$ item deletions and $M \times N$ item insertions for the e-MAT, leading to $2 \times M \times N$ switch-OS-based plane interactions. As we introduce in Subsection III-B, switch-OS-based plane interactions are time-consuming, so reconstructing the e-MAT structure cannot achieve fast replacement. Our updating mechanism aims to reduce switch-OS-based plane interactions while maintaining functional correctness. The key

TABLE I
SYMBOLS USED IN THE E-MAT UPDATING MECHANISM.

O_new	The non-offloaded object that needs offloading.
O_old	The offloaded object that needs replacing.
MAT_m	The m-th MAT in the e-MAT structure.
I_new_m	The item for O_new in MAT_m.
I_old_m	The item for O_old in MAT_m.

idea of our updating mechanism is to keep all merged items in the e-MAT structure unchangeable while replacing the target offloaded object. To provide a clear and concise description of our updating mechanism, in the following content, we first define the symbols used, as shown in Table I, and then provide the details of the mechanism.

Our updating mechanism sequentially examines each MAT to determine if its items need modification. Specifically, for MAT_m, we categorize the situations into the following cases:

Case 1. $I\_new_m = I\_old_m$. In this case, we don't modify any item in the MAT_m because deleting I_old_m and inserting I_new_m cancel each other.

Case 2. $I\_new_m \neq I\_old_m$. In this case, we delete I_old_m only if it isn't shared by other offloaded objects; similarly, we insert I_new_m only if it doesn't exist in MAT_m.

We use the following examples to illustrate how the e-MAT updating mechanism works.

Example 1. Suppose we want to replace O_old $(0x111)$ in Figure 8 with O_new $(0x114)$, we can find that for MAT_1, there is $I\_old_1 = I\_new_1 = 0x1$, which fits Case 1, so we don't need to modify MAT_1. Similarly, for MAT_2, there is $I\_old_2 = I\_new_2 = \{0x1, 0x1\}$, which also fits Case 1, so we don't need to modify MAT_2. Finally, for MAT_3, there are $I\_old_3 = \{0x1, 0x1\}$ and $I\_new_3 = \{0x1, 0x4\}$, which fits Case 2. Since I_old_3 isn't shared by other offloaded objects and I_new_3 doesn't already exist in MAT_3, we can safely delete I_old_3 from MAT_3 and insert I_new_3 to MAT_3.

Example 2. Suppose we want to replace O_old $(0x211)$ in Figure 8 with O_new $(0x114)$, we can find that for MAT_1, there is I_old_1 $(0x2)$ $\neq$ I_new_1 $(0x1)$, which fits Case 2. Since I_old_1 isn't shared by other offloaded objects but I_new_1 already exists in MAT_1, we only need to delete I_old_1 from MAT_1. Similarly, for MAT_2, there is I_old_2 $(\{0x2, 0x1\})$ $\neq$ I_new_2 $(\{0x1, 0x1\})$, which fits Case 2. Since I_old_2 isn't shared by other offloaded objects but I_new_2 already exists in MAT_2, we only need to delete I_old_2 from MAT_2. Finally, for MAT_3, there are $I\_old_3 = \{0x2, 0x1\}$ and $I\_new_3 = \{0x1, 0x4\}$, which fits Case 2. Since I_old_3 isn't shared by other offloaded objects and I_new_3 doesn't already exist in MAT_3, we can safely delete I_old_3 from MAT_3 and insert I_new_3 to MAT_3.

Due to the fine-grained operations in different cases, our e-MAT updating mechanism needs no more than $2 \times M$ switch-OS-based plane interactions to update the e-MAT. Besides, it keeps all merged items in the e-MAT structure unchangeable while replacing the target offloaded object. This guarantees the functional correctness of the e-MAT.

C. Switch-server Co-offloading Mechanism

To achieve fast and precise offloading, we propose a switch-server co-offloading mechanism. The key idea of our mechanism is to achieve precise offloading by using both the switch data plane and backend servers to collect accurate statistics – rather than relying solely on the switch data plane – and to achieve fast offloading with the real-time min-heap, the hybrid plane interactions, and the batch offloading optimization. In this subsection, we first describe the offloading process, and then present the technical details about how to achieve fast and precise offloading.

Offloading Process. Figure 8 shows Hare's hotspot offloading process which can be divided into the following steps: Step 1, the OO-Stat counts the frequency for each offloaded object. In the meantime, the NOO-Stat counts the frequency for each non-offloaded object. Step 2, the analyzer retrieves frequency counts from the OO-Stat to identify a set O_{switch} of K coldest offloaded objects. Then, the analyzer retrieves K hottest non-offloaded objects from each backend server, and from these retrieved objects, the analyzer identifies a set O_{server} consisting of the overall K hottest non-offloaded objects. Step 3, the analyzer identifies a set $O_{hottest}$ of K globally hottest objects (whether offloaded or not) among O_{switch} and O_{server}, and then decides the offloading policy. To be specific, objects which belong to O_{switch} but don't belong to $O_{hottest}$ should be replaced by objects which belong to both O_{server} and $O_{hottest}$. Step 4, the analyzer modifies the object states in the state managers for objects participating in the replacement. Step 5, the analyzer notifies the M-modifier[6] to modify the hit-check e-MAT by APIs, while also notifying the D-modifier[7] on each backend node to modify the application data. Step 6, the analyzer refreshes the states of objects participating in the offloading policy. Step 7, the analyzer clears the OO-Stat and informs the NOO-Stat on each backend server to clear itself.

How to Enable Fast and Precise Offloading. While it's straightforward to achieve precise offloading by accurately identifying the hottest non-offloaded objects and the coldest offloaded objects with precise counters, the challenge is how to speed up the offloading process. Specifically, the following procedures may introduce significant time overhead:

Proc 1. The NOO-Stat on each backend server needs to prepare a set of top-K hottest non-offloaded objects in Step 2. However, accurately identifying these objects by scanning the frequency counts of all non-offloaded objects can be a time-intensive process.

Proc 2. The analyzer on the switch control plane needs to retrieve all frequency counts from the OO-Stat on the switch data plane in Step 3, leading to a high volume of plane interactions. However, as discussed in Subsection III-B, switch-OS-based plane interactions are inherently time-consuming.

Proc 3. Both the OO-Stat and NOO-Stat require substantial time to get a large volume of statistics in Step 1. This ensures the statistical data aligns closely with the workload distribution, but it also results in a significant time cost.

[6] M-modifier is short for the hit-check e-MAT modifier.

[7] D-modifier is short for the application data modifier.

We address the impact of these procedures on overall offloading latency using the following techniques:

Tech 1. Real-time Min-heap. We make the NOO-Stat in each backend server maintain a real-time min-heap of the objects with the top-K highest frequency counts while collecting frequency counts in Step 1. Specifically, when a query comes to the backend server, if the queried object isn't in the min-heap but the frequency count of the queried object is higher than the frequency count of the object at the heap top, the min-heap will delete the object at the heap top and insert the queried object into the heap. In this way, after Step 1 is completed, the NOO-Stat can directly identify the top-K hottest non-offloaded objects with the min-heap.

Tech 2.Hybrid Plane Interaction. We realize e-MAT modifications using switch-OS-based APIs while realizing other plane interactions using network packets. To support network-based plane interaction, the OO-Stat uses the register to realize the precise counter, and the state manager also uses the register to store the object state. Because the rate of network packets can significantly exceed the rate of switch-OS-based plane interactions by several orders of magnitude, operations such as retrieving frequency counts from the OO-Stat, which are achievable through network-based plane interactions, are no longer time-consuming.

Tech 3. Custom Batch Offloading. As detailed in the paragraph titled "Offloading process", Hare's offloading process supports batch offloading, which allows offloading K hottest objects in a single execution of the offloading process (Step 1 $\sim$ Step 7), instead of offloading one object at a time. By offloading K hottest objects together with the statistics collected once in Step 1, batch offloading reduces the average time required to offload each object.

Equation 1 illustrates the average time (T_{avg}) required to offload a single object when using the three techniques above.

$$T_{avg} = \frac{\sum_{step=1}^{7} T_{step}}{K} \approx \frac{T_1 + T_5}{K} \approx \frac{T_{stat} + T_{MAT}}{K} \quad (1)$$

As shown in Equation 1, among all steps in the offloading process, Step 1 and Step 5 are the most time-consuming. In Step 1, the time (T_{stat}) for both the OO-Stat and the NOO-stat to collect statistics needs milliseconds to get a large number of statistics so that the distribution of the statistics is similar enough to the distribution of the workload. In Step 5, the time (T_{MAT}) spent on MAT modification is also significant because the M-modifier realizes MAT modification with slow switch-OS-based APIs, and the number of times for calling the APIs is in proportion to K. As a result, increasing K can mainly decrease the impact of T_{stat} on T_{avg} but hardly ease the impact of T_{MAT}. What's more, a large K increases the average time the NOO-Stat spends on collecting statistics (counting frequency & updating the min-heap) for each query received, because the time complex for NOO-Stat to update its min-heap is $O(logK)$. Since the query-processing function and the NOO-Stat's statistic-collecting function are sequentially executed for each query received, a large K causes a large interval between two invocations of the query processing function, which decreases the backend server throughput. So, when choosing the value of K, there is a balance between the average time for offloading one object and the average time the NOO-Stat takes to collect statistics.

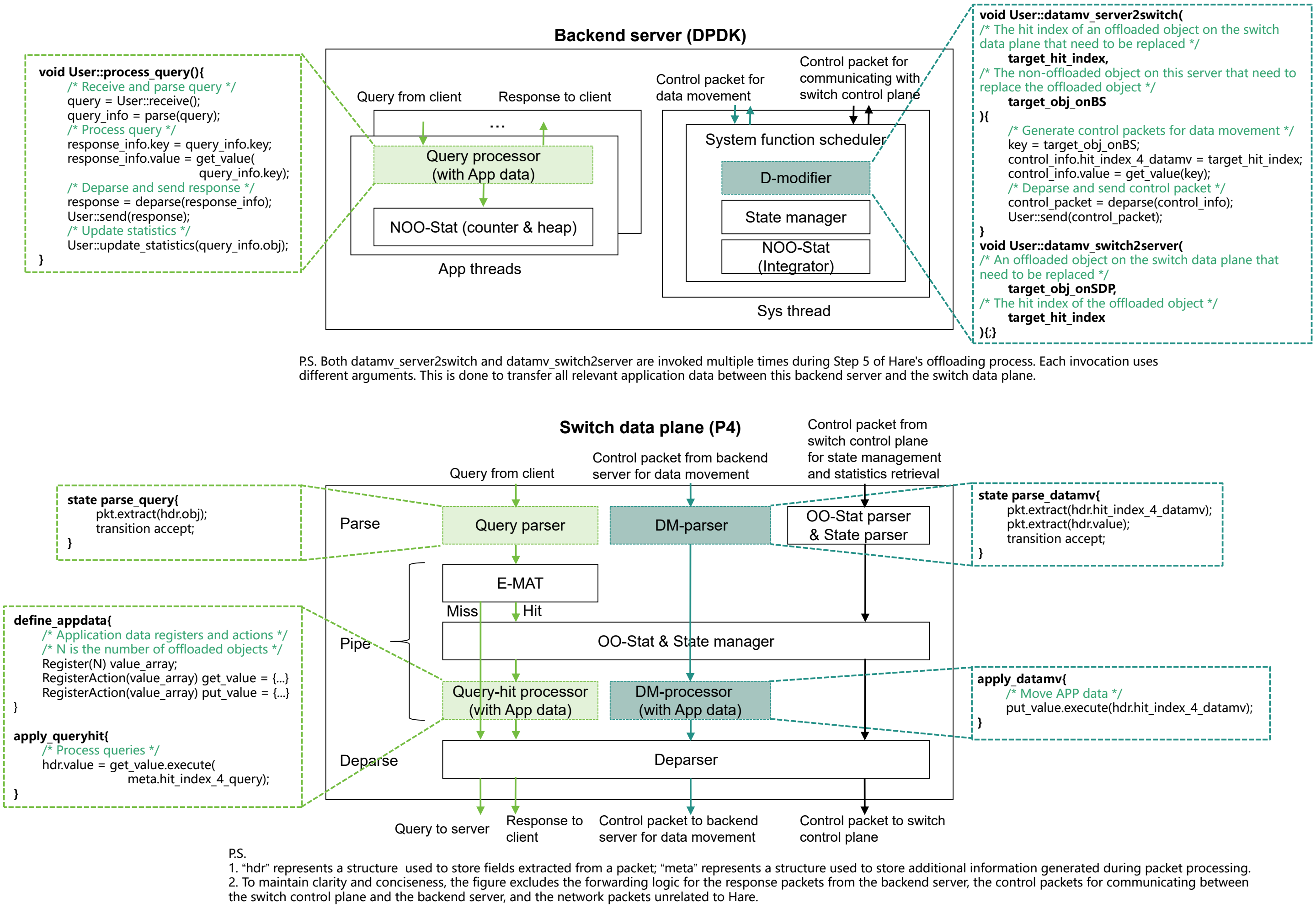
D. Automatic Offloading Mechanism

To facilitate application development, Hare introduces a programming framework that abstracts the complexities of dynamic hotspot offloading. In this subsection, we first introduce the challenge in framework design. Next, we present the key idea behind the design. Finally, we describe the user interfaces and the executable file auto-generation mechanism.

Challenge. It's challenging to abstract a framework for hotspot offloading that can flexibly accommodate various applications, as the hotspot offloading logic is highly intertwined with user-defined application-related query processing logic. For instance, to avoid data inconsistency, the application data should be appropriately transferred between the backend servers and the switch data plane in Step 5 of Hare's hotspot offloading process. However, the structure (e.g., map or queue) of the application data is closely tied to the specific application. As a result, transferring the application data uniformly across different applications may cause incorrect application behavior.

Key Idea. The key idea of the programming framework is to hide the implementation of application-unrelated functions from users, while providing users with only application-related programming interfaces. In this way, the framework can reduce development difficulty while maintaining application flexibility. Figure 8 illustrates how to integrate a user-defined application into Hare's dynamic hotspot offloading process with the programming framework. With Hare, the user needs to complete the following tasks: First, write code related to application-specific query processing and data movement in the P4 file for the switch data plane. Second, write code related to application-specific query processing and data movement in the DPDK file for the backend server. Third, run a Python tool, which takes the user's codes and configuration parameters (e.g., the number of objects offloaded) as input, to generate complete code files and executable files. Finally, run executable files on appropriate devices to launch the system.

User Interfaces for Backend Server. We first introduce the components on the backend server and then describe the user interfaces with an example. As shown in Figure 11, the components on the backend server include the query processor, the NOO-Stat, the D-modifier, the state manager, and the system function scheduler. The NOO-Stat further consists of counters, heap(s), and an integrator, which are distributed across different threads – the thread running the query processor (called App thread) is equipped with counters and a K-sized min-heap for queried objects; the thread running the system function scheduler (called Sys thread) is equipped with an integrator, which is used to find the K hottest objects from all heaps of the backend server when there is more than one App thread. The system function scheduler manages when and which component of the Sys thread should run according to the hotspot offloading process mentioned in Subsection IV-C. Among the components above, only the query processor and the D-modifier remain user-defined. The former is for flexible application-related query processing,



```
void User::process_query(){
        /* Receive and parse query */
        query = User::receive();
        query_info = parse(query);
        /* Process query */
        response_info.key = query_info.key;
        response_info.value = get_value(
                                        query_info.key);
        /* Deparse and send response */
        response = deparse(response_info);
        User::send(response);
        /* Update statistics */
        User::update_statistics(query_info.obj);
}
```

```
void User::datamv_server2switch(
/* The hit index of an offloaded object on the switch
data plane that need to be replaced */
        target_hit_index,
/* The non-offloaded object on this server that need to
replace the offloaded object */
        target_obj_onBS
){
        /* Generate control packets for data movement */
        key = target_obj_onBS;
        control_info.hit_index_4_datamv = target_hit_index;
        control_info.value = get_value(key);
        /* Deparse and send control packet */
        control_packet = deparse(control_info);
        User::send(control_packet);
}
void User::datamv_switch2server(
/* An offloaded object on the switch data plane that
need to be replaced */
        target_obj_onSDP,
/* The hit index of the offloaded object */
        target_hit_index
){;}
```

P.S. Both datamv_server2switch and datamv_switch2server are invoked multiple times during Step 5 of Hare's offloading process. Each invocation uses different arguments. This is done to transfer all relevant application data between this backend server and the switch data plane.

```
state parse_query{
        pkt.extract(hdr.obj);
        transition accept;
}
```

```
define_appdata{
        /* Application data registers and actions */
        /* N is the number of offloaded objects */
        Register(N) value_array;
        RegisterAction(value_array) get_value = {...}
        RegisterAction(value_array) put_value = {...}
}

apply_queryhit{
        /* Process queries */
        hdr.value = get_value.execute(
                                meta.hit_index_4_query);
}
```

```
state parse_datamv{
        pkt.extract(hdr.hit_index_4_datamv);
        pkt.extract(hdr.value);
        transition accept;
}
```

```
apply_datamv{
        /* Move APP data */
        put_value.execute(hdr.hit_index_4_datamv);
}
```

P.S.
1. "hdr" represents a structure used to store fields extracted from a packet; "meta" represents a structure used to store additional information generated during packet processing.
2. To maintain clarity and conciseness, the figure excludes the forwarding logic for the response packets from the backend server, the control packets for communicating between the switch control plane and the backend server, and the network packets unrelated to Hare.

Fig. 11. User Interfaces. Components with dashed boxes are defined by user, while the others are defined by the framework.

while the latter is to facilitate appropriate application data movement. We keep other components hidden from users since they are application-unrelated and can be automatically handled by the framework.

We take the read-only KVS system as an example, as shown in Figure 11. The bold code in dashed boxes is provided by the framework and the non-bold code in dashed boxes is provided by the user. We can see that the user is required to implement three application-related member functions (process_query, datamv_server2switch, and datamv_switch2server) of the User class. When implementing the functions above, the user can also utilize other built-in member functions (e.g., send, receive, update_statistic) of the User class. Specifically, to implement a read-only KVS system, in process_query function, first, the user receives and parses the query to get the queried key. After that, the user finds the corresponding value of the queried key, constructs the response packet, and sends the packet to the client. Finally, the user updates the NOO-Stats with the queried key. In datamv_server2switch function, the user constructs and sends the control packet for application data movement to the switch data plane. Each control packet contains the hit index of an offloaded key that should be replaced (control_info.hit_index_4_datamv) and the value of a non-offloaded key that should be offloaded to the switch data plane (control_info.value). In datamv_switch2server function, because the switch in a read-only KVS system doesn't have dirty data, there is no need for programming. If other systems, like lock manager systems, require application data movement from the switch data plane to the backend servers, the user can send and receive control packets in this function to retrieve application data from the switch data plane.

User Interfaces for Switch Data Plane. We first introduce the components on the switch data plane and then describe the user interfaces with an example. As shown in Figure 11, the components on the switch data plane include parsers, the e-MAT structure, the OO-Stat, the state manager, packet processors, and the deparser. Among them, parsers can be further divided into four types: the query parser, the DM-parser[8], the OO-Stat parser, and the state parser; similarly, packet processors can be divided into two types: the query-hit processor and the DM-processor[9]. To reduce development difficulty, we keep the OO-Stat parser, the state parser, the E-MAT, the OO-Stat, the state manager, and the deparser hidden

[8] DM-parser is short for the parser used to parse the control packet for application data movement. Similarly, the OO-Stat parser is short for the parser used to parse the control packet for fetching and cleaning statistics; the state parser is short for the parser used to parse the control packet for state management.

[9] DM-processor is short for the processor used to process the control packet for application data movement.

from users. These components are application-unrelated and can be automatically handled by the framework. The query parser and the query-hit processor remain user-defined to enable flexible application-related query processing; the DM-parser and the DM-processor are user-defined as well to facilitate appropriate application data movement.

We take the read-only KVS system as an example, as shown in Figure 11. The bold code in dashed boxes is provided by the framework and the non-bold code in dashed boxes is provided by the user. We can see that the user is required to implement five application-related code regions: parse_query, define_appdata, apply_queryhit, parse_datamv, and apply_datamv. The former three are used for query processing, while the latter two are used for application data movement. To enable query processing, the user first extracts an object from the appropriate position of the query packet in the parse_query region. Next, the user defines the register array (e.g., value_array) and the related register actions (e.g., get_value and put_value) to support the operations on values for the KVS system in the define_appdata region. Finally, the user processes the hit query with the register array, the related register actions, and the hit index in the apply_queryhit region. Similarly, to enable application data movement, the user first extracts the hit index and the value from the control packet, which originates from the backend server for data movement, within the parse_datamv region. Then, in the apply_datamv region, the user puts the value into the correct position of the register array based on the extracted hit index.

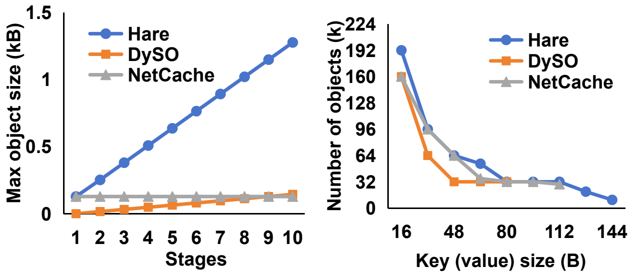
The framework automatically drops queries for objects participating in replacement to maintain functional correctness. The states of objects participating in replacement are special, and queries for objects with special states will be dropped by the framework instead of being exposed to the user. The state modification is managed by the analyzer in Step 4 and Step 6 of the switch-server co-offloading mechanism.

Executable File Auto-generation Mechanism. A Python tool is designed to auto-generate the complete compilable code files based on the user's code files and the user's configuration parameters (e.g., the number of offloaded objects). The code file for the switch control plane is unrelated to query processing so it's purely generated by the Python tool without any user's code file. After generating compilable code files, the Python tool calls corresponding compilers for different code files to generate the executable files. Additionally, for the backend server, hotspot-offloading-related components can be provided via a library, enabling the user to integrate Hare's hotspot offloading logic into existing DPDK frameworks.

V. Evaluation

A. Experimental Setup

Testbed. Our testbed consists of one 3.2Tbps Barefoot Tofino switch and eight server machines. Each server machine is equipped with two 12-core CPUs (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz), 256 GB total memory (eight Samsung 32GB DDR4-2666 memory), and a 100G NIC (Mellanox Technologies MT27800 Family [ConnectX-5]). Six servers are used as clients to generate queries and two servers are used as backend nodes to respond to queries for non-offloaded objects.



(a) Maximum hotspot size vs. stages occupied.

(b) Number of hotspots vs. key (value) size.

Fig. 12. Flexibility in hotspot size.

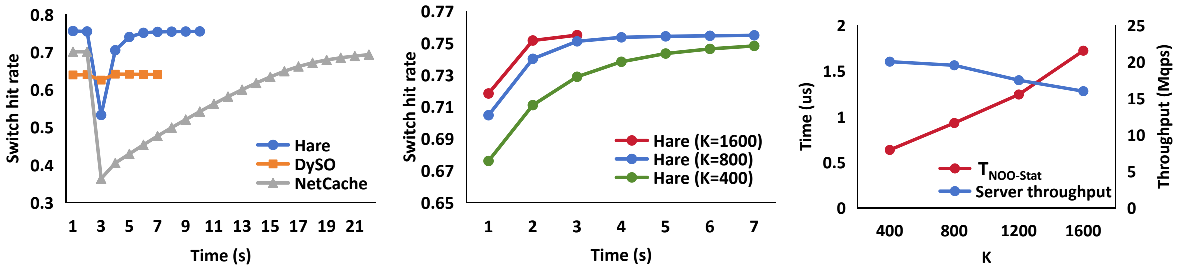
Applications. We cover both read-intensive and write-intensive applications for Hare [45]. As we introduce in Section I, NetCache is designed for the read-intensive KVS system, DySO is designed for read-only systems, and Hare is designed as a systematic framework for general systems. The intersection of the systems all three designs support is the read-only KVS system, so we compare Hare with NetCache and DySO within this context. By default, for Hare, we set $T_{wait} = 50ms$, $K = 800$ to gain the best system performance. For NetCache, we set the threshold for hotspot report to 384, sample 128 frequency counts from the switch data plane to find the coldest offloaded objects, and clean statistics every 1s. For Hare, we use 1Mpps control packets to fetch records from the switch data plane to the switch control plane and half frequency counts every 1s. We also implement a lock manager system with Hare to show Hare suits write-intensive applications as well. Our implementation is similar to NetLock [18]. NetLock implements variable-length query-waiting queues for offloaded locks but only supports static hotspot offloading in the initial phase. We implement queues with a fixed size of 8 to simplify the movement of application data.

Workloads. For both applications, we use skewed workloads that follow Zipf distribution with skewness parameter $\alpha = 0.99$, which are typical for data center scenarios [17], [42], [46] and are evidenced by real-world deployments [27], [47].

B. Effect of Individual Optimization

Flexibility in Hotspot Size. Figure 12a compares the maximum object size that Hare, DySO, and NetCache can support with a certain number of stages on the switch data plane. We observe that Hare enables large maximum object size compared with DySO and NetCache. Specifically, when 10 stages are used to store offloaded objects, Hare supports 8.86× larger object size than DySO and 9.97× larger than NetCache, because Hare's e-MAT structure allows each stage to store and perform hit-checks for maximally 128B object segments, while DySO allows for maximally 16B object segments for each stage. There is no intuitive approach for NetCache to extend the maximum object size as described in Section III.

We also validate that Hare doesn't sacrifice the number of offloaded objects. We test the maximum number of objects (keys) that can be offloaded under different object (key) sizes and application data (value) sizes, as shown in Figure 12b. We observe that Hare always supports the largest number of offloaded objects. Specifically, Hare supports $1\times \sim 2\times$



(a) Switch hit rate vs. time. (b) The impact of K in switch hit rate. (c) The impact of K in server throughput.

Fig. 13. Switch hit rate recovery capability.

more objects than DySO and $1\times \sim 1.5\times$ more objects than NetCache when the object size and the application data size range from 16B to 80B. It's because compared with NetCache, Hare moves the NOO-Stat from the switch data plane to the backend servers, while the NOO-Stat in NetCache occupies registers in two stages of the switch data plane. Compared with DySO which uses the register-based r-MAT structure to store objects, Hare benefits from using the MAT-based e-MAT structure to store objects, as the MAT space is nearly $1.8\times$ larger than the register space in one stage.

Switch Hit Rate Recovery Capability. We validate that Hare offloads hotspots fast and precisely, in terms of recovery curves of the switch hit rate. We test Hare, DySO, and NetCache with 80B keys and 80B values, because three systems offload the same number of hotspots (32k) to the switch data plane under this configuration. All systems start with zero offloaded objects and warm for 100s to achieve a stable switch hit rate. Then we switch 32k coldest objects to the top of the popularity ranks to test the recovery capability of the three systems. Figure 13a illustrates how the switch hit rate changes over time. We make the following observations. First, Hare achieves the highest recovered switch hit rate. Specifically, its recovered switch hit rate outperforms NetCache by 9% and outperforms DySO by 17%. It's because NetCache utilizes space-saving algorithms (Count-Min sketch and Bloom filter) on the switch data plane to report hotspots. However, this approach loses precision in finding the most popular 32k objects. DySO's recovered switch hit rate is limited by the number of columns of its r-MAT structure. When only one column is feasible for large hotspots, severe hash collisions restrict DySO's recovered switch hit rate. In contrast, Hare counts the frequency for all objects precisely and offloads objects without severe hash collision, so Hare can reach the highest recovered switch hit rate. Second, Hare requires less time to achieve the same switch hit rate than NetCache. Specifically, to reach NetCache's recovered switch hit, Hare needs 10% of the total times required by NetCache, because NetCache wastes time on switch-OS-based plane interactions while Hare realizes most plane interactions with efficient network packets and uses the custom batching offloading method to reduce the average time cost. Notably, DySO sacrifices its final recovered switch hit rate to achieve the fastest offloading with its register-based r-MAT structure.

We also test the impact of batch size (K) on Hare's switch hit rate as shown in Figure 13b. We observe that a large K tends to reduce the switch hit rate recovery time, and the reduction becomes marginal as K continues to increase. It's because when K increases, the time bottleneck for hotspot offloading changes from collecting statistics (Step 1) to modifying MATs (Step 5), while the latter hardly benefits from a large K as we analyze in Section IV-C. What's more, we evaluate the impact of batch size on the backend server throughput as shown in Figure 13c. We make the following observations. First, when K increases, the average time ($T_{NOO-Stat}$) increases, which is required by the NOO-Stat to collect statistics (counting frequency & updating the min heap) from each received query that triggers min heap updating. It's because the time complexity for the NOO-Stat to update its min heap is $O(logK)$. Second, the backend server throughput decreases when K increases, because each received query triggers a sequential execution of query-processing and statistic-collecting. A large K causes a large interval between two query processing, which lowers the backend server throughput.

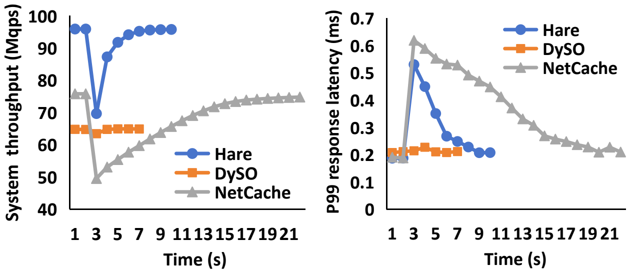
TABLE II
LoC Comparison (K).

	KVS			Lock manager	
	NetCache	DySO	Hare	NetLock	Hare
Backend server	1.3	1.2	0.2	1.9	0.7
Switch control plane	1.7	2.2	0	0.7	0
Switch data plane	1.2	1.1	0.2	2.7	2.0
Total	4.2	4.5	0.4	5.3	2.7

High Programmability. Table II illustrates the lines of user-written code (LoC) of Hare with the existing arts of the read-only KVS system and the lock manager system. We observe that Hare consistently requires the fewest LoCs. It's because for both the backend server and the switch data plane, Hare abstracts hotspot offloading functions and basic networking tasks (e.g., sending, receiving, and header processing). Moreover, with Hare, the switch control plane requires no LoC, as its application-unrelated functions are entirely auto-generated.

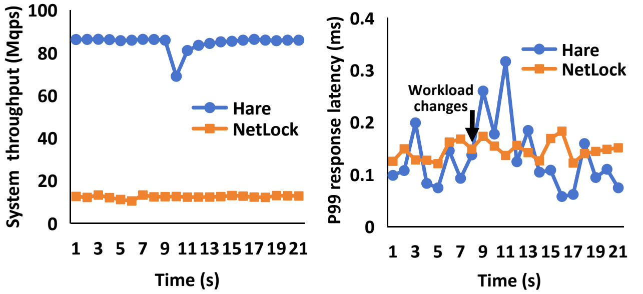
C. End-to-end Performance

For the KVS application, we test the system throughput and the P99 query response latency of Hare, DySO, and NetCache with 80B keys and 80B values. For the lock manager application, we test the system throughput and the P99 query response latency of Hare and NetLock. All systems are allowed to offload up to 32k objects to the switch data plane. Before workload changes, all systems start with zero offloaded objects and warm for 100s to achieve stable system throughput and P99 query response latency. For each system, we fine-tune the client query rate to a specific value, ensuring that before workload changes, the stable P99 query response



(a) System throughput vs. time. (b) P99 response latency vs. time.

Fig. 14. The performance of the KVS systems.



(a) System throughput vs. time. (b) P99 response latency vs. time.

Fig. 15. The performance of the lock manager systems.

latency remains within a predetermined upper bound and the system throughput is maximized. Then we make the workload change by moving 32k coldest objects to the top of the popularity ranks to test the recovery capability of the systems.

KVS. Figure 14 shows the recovery curves of the system throughput and the P99 query response latency when the stable P99 query response latency $\leq$ 200us. Similar conclusions hold for other latency constraints (e.g., 400us and 800us). We make the following observations. First, Hare achieves the highest recovered system throughput. Specifically, Hare achieves 1.27× higher recovered system throughput than that of NetCache and 1.48× higher than that of DySO. It's because Hare achieves the highest recovered switch hit rate as shown in Figure 13a. A high switch hit rate can relieve the query-dropping issue that happens on backend servers and reduce the P99 query response latency, which enables Hare to achieve high system throughput under given latency constraints. Second, compared to NetCache, Hare requires less time to recover the system throughput and the P99 query response latency to a stable value. Specifically, Hare needs around 40% of the total time of NetCache to recover the system throughput and the P99 query response latency. It's because the recovery time mainly depends on the switch hit rate recovery time, and Hare needs around 40% of the total time required by NetCache to recover the switch hit rate to a stable value.

Lock Manager. Figure 15 shows the recovery curves of the system throughput and the P99 query response latency when the stable P99 query response latency $\leq$ 200us. We observe that Hare achieves 6.61× higher throughput than NetLock and can fast recover the system throughput and the P99 response latency within 8s. It's because Hare supports fast and precisely dynamic hotspot offloading, whereas NetLock only supports static hotspot offloading during the initial phase. As a result, when facing dynamic workloads, the system performance of NetLock decreases to the level where there is no offloading.

In a word, Hare achieves the highest system throughput with a given P99 query response latency when the switch hit rate stabilizes. It can also fast recover the system throughput and the P99 query response latency when the workload changes. Because of these two traits, Hare can help latency-intensive applications retain users and increase profits.

VI. Related Work

To our knowledge, this is the first paper to propose a systematic framework that provides efficient and generally automatic hotspot offloading on programmable switches. We describe other related works in the following aspects:

Applications Benefiting from Hotspot Offloading. In addition to the KVS and the lock manager, there are other applications that have the potential to benefit from hotspot offloading, such as network address translation (NAT) boxes [48], online transaction processing (OLTP) [19], L4 load balancers [49], and content delivery networks (CDNs) [50]–[52]. All of these applications share the common characteristic of experiencing frequent workload changes in their environments.

Switch-based Hotspot Offloading Mechanisms. Besides NetCache [17] and DySO [20], there are still other works about switch-based hotspot offloading, while each of them has its own limitation. Specifically, NetLock [18] and P4DB [19] only support static hotspot offloading in the initial phase; PFCA [53]'s dynamic hotspot offloading process is only simulation-viable [54], [55]; NetHCF [56] dynamically offloads hotspots in a manner highly similar to NetCache, which wastes time on slow switch-OS-based plane interactions. What's more, all the above works are tailored to specific applications. On the contrary, Hare offloads hotspots not only dynamically but also fast and precisely. Besides, it further supports high flexibility in hotspot sizes and provides a systematic framework for generally automatic hotspot offloading.

Fast Plane Interaction. Existing works [20], [57]–[61] realize that the heavy intervention of the switch-OS for plane interactions causes additional overhead and should be avoided or optimized, however, only a small number of works make an effort on it. Specifically, DySO [20] achieves fast plane interactions by network packets; IMap [58] and Symposium [59] double MAT/register-based modules on the switch data plane to overlap heterogeneous operations like reading and writing while these operations are still based on switch-OS-based APIs; Mantis [61] modifies the existing switch-OS drivers and control plane interfaces. These optimization methods either cause heavy resource occupation on the switch data plane or introduce driver-level intrusion which may result in system instability. In contrast, Hare relies on both the switch-OS-based APIs and network packets to achieve fast plane interactions, easing resource occupation and avoiding OS intrusion.

Automatic Tools for Application Offloading. There are other automatic tools for application offloading on smart network devices. [62]–[66] provides tools to automatically generate high-performance codes for network function accelerators (e.g., FPGAs, P4 programmable switches) from existing unaccelerated code through code analysis and performance profiling. [67] and [68] provide function correctness validation for applications on the switch data plane. These aforementioned automatic tools primarily focus on application migration, performance monitoring, and performance analysis for individual devices. In contrast, Hare is a programming framework designed to hide complex interactions between the switch and the backend servers when users develop systems that require dynamic hotspot offloading.

VII. Conclusion

Hare is the first systematic framework for efficient and generally automatic hotspot offloading on programmable switches. It has high flexibility in hotspot size, offloads hotspots fast and precisely, and provides an offloading-invisible development process. We compare Hare with NetCache and DySO in a read-only KVS system, and with NetLock in a lock manager system. The experimental results demonstrate that our design supports the largest hotspot size, requires the fewest LoCs, and rapidly achieves the highest switch hit rate and the highest system throughput with narrow peaks in P99 query response latency when workloads change.

Acknowledgments

The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2022ZD0119301), and the National Natural Science Foundation of China under the grant number (62472384, U24A20326). Zeke Wang is the corresponding author.

References

[1] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," *SIGARCH Comput. Archit. News*, 2015.

[2] J. Hamilton. (2009) The cost of latency. http://perspectives.mvdirona.com/2009/10/31/-TheCostOfLatency.aspx.

[3] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and http chunking in web search," in *Velocity*, 2009.

[4] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, 2013.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007.

[6] Y. He, S. Elnikety, J. Larus, and C. Yan, "Zeta: Scheduling interactive services with partial execution," in *SoCC*, 2012.

[7] J. Yi, F. Maghoul, and J. Pedersen, "Deciphering mobile search patterns: A study of yahoo! mobile search queries," in *WWW*, 2008.

[8] Z. Wang, H. Li, L. Sun, T. Rosenkrantz, H. Che, and H. Jiang, "Tailguard: Tail latency slo guaranteed task scheduling for data-intensive user-facing applications," in *ICDCS*, 2023.

[9] Storage: How Tail Latency Impacts Customer-Facing Applications. https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications.

[10] AWS. (2019) Annual report. https://s2.q4cdn.com/299287126/files/doc_financials/2020/ar/2019-Annual-Report.pdf.

[11] R. Han, J. Wang, S. Huang, C. Shao, S. Zhan, J. Zhan, and J. L. Vazquez-Poletti, "Pcs: Predictive component-level scheduling for reducing tail latency in cloud online services," in *ICPP*, 2015.

[12] G. Zhang, K. Ren, J.-S. Ahn, and S. Ben-Romdhane, "Grit: Consistent distributed transactions across polyglot microservices with multiple databases," in *ICDE*, 2019.

[13] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *ASPLOS*, 2019.

[14] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martínez, "Workload characterization of interactive cloud services on big and small server platforms," in *IISWC*, 2017.

[15] S. Chen, Y. Jiang, C. Delimitrou, and J. F. Martínez, "Pimcloud: Qos-aware resource management of latency-critical applications in clouds with processing-in-memory," in *HPCA*, 2022.

[16] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for second-scale tail latency," in *NSDI*, 2019.

[17] 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 *SOSP*, 2017.

[18] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "Netlock: Fast, centralized lock management using programmable switches," in *SIGCOMM*, 2020.

[19] M. Jasny, L. Thostrup, T. Ziegler, and C. Binnig, "P4db - the case for in-network oltp," in *SIGMOD*, 2022.

[20] C. H. Song, X. Z. Khooi, D. M. Divakaran, and M. C. Chan, "Revisiting application offloads on programmable switches," in *IFIP Networking*, 2022.

[21] Z. István, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *FPL*, 2013.

[22] J. Liu, A. Dragojević, S. Fleming, A. Katsarakis, D. Korolija, I. Zablotchi, H.-C. Ng, A. Kalia, and M. Castro, "Honeycomb: Ordered key-value store acceleration on an fpga-based smartnic," *IEEE Transactions on Computers*, 2024.

[23] A. Conway, A. Gupta, V. Chidambaran, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson, "Splinterdb: Closing the bandwidth gap for nvme key-value stores," in *ATC*, 2020.

[24] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites," in *WWW*, 2002.

[25] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *HotNets*, 2014.

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

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

[28] J. Zhang, S. Cheng, Z. Xue, J. Deng, C. Fu, W. Zhou, S. Wang, C. Chen, and F. Li, "Esdb: Processing extremely skewed workloads in real-time," in *SIGMOD*, 2022.

[29] 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 *FAST*, 2020.

[30] "Intel (r) programmable ethernet switch products," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, 2021.

[31] "Cavium-xpliant (r) family of programmable ethernet switches," https://www.openswitch.net/cavium/, 2021.

[32] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *SIGCOMM*, 2007.

[33] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, 2015.

[34] 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," in *SIGCOMM*, 2013.

[35] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, 2014.

[36] Y. Zhan, Y. Jiao, D. E. Porter, A. Conway, E. Knorr, M. Farach-Colton, M. A. Bender, J. Yuan, W. Jannen, and R. Johnson, "Efficient directory mutations in a full-path-indexed file system," *ACM Trans. Storage*, 2018.

[37] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, 2010.

[38] Apache. (2023) Iotdb user guide. https://iotdb.apache.org/zh/UserGuide/latest/Basic-Concept/Data-Model-and-Terminology.html.

[39] K. Mohamed and D. Wijesekera, "A lightweight framework for web services implementations on mobile devices," in *MobiSys*, 2012.

[40] Y. Chow, W. Zhu, C.-L. Wang, and F. Lau, "State-on-demand execution for adaptive component-based mobile agent systems," in *ICPADS*, 2004.

[41] NoviFlow. (2017) Noviflow noviswitch. http://noviflow.com/products/noviswitch/.

[42] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with switchkv," in *NSDI*, 2016.

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

[44] A. Z. Broder and M. Mitzenmacher, “Network applications of bloom filters: A survey,” *Internet Mathematics*, 2004.

[45] Hare. [Online]. Available: https://github.com/LaceFern/Hare.git

[46] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, “EC-Cache: Load-Balanced, Low-Latency cluster caching with online erasure coding,” in *OSDI*, 2016.

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

[48] D. Kim, Y. Zhu, Z. Liu, C. Kim, J. Lee, V. Sekar, and S. Seshan, “Tea: Enabling state-intensive network functions on programmable switches,” in *SIGCOMM*, 2020.

[49] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo, “Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing,” in *NSDI*, 2022.

[50] C. Wernecke, H. Parzyjegla, G. Mühl, E. Schweissguth, and D. Timmermann, “Flexible notification forwarding for content-based publish/subscribe using p4,” in *NFV-SDN*, 2019.

[51] C. Wernecke, H. Parzyjegla, G. Mühl, P. Danielis, and D. Timmermann, “Realizing content-based publish/subscribe with p4,” in *NFV-SDN*, 2018.

[52] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé, “Packet subscriptions for programmable asics,” in *HotNets*, 2018.

[53] G. Grigoryan, Y. Liu, and M. Kwon, “Pfca: A programmable fib caching architecture,” *IEEE/ACM Transactions on Networking*, 2020.

[54] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, “Heavy-hitter detection entirely in the data plane,” in *SOSR*, 2017.

[55] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, “Efficient measurement on programmable switches using probabilistic recirculation,” in *ICNP*, 2018.

[56] M. Zhang, G. Li, X. Kong, C. Liu, M. Xu, G. Gu, and J. Wu, “Nethcf: Filtering spoofed ip traffic with programmable switches,” *IEEE Transactions on Dependable and Secure Computing*, 2023.

[57] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, “An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends,” *IEEE Access*, 2021.

[58] G. Li, M. Zhang, C. Guo, H. Bao, M. Xu, and H. Hu, “Switches are scanners too! a fast and scalable in-network scanner with programmable switches,” in *HotNets*, 2021.

[59] H. Namkung, D. Kim, Z. Liu, V. SekaR, and P. Steenkiste, “Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations,” in *SOSR*, 2021.

[60] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, “SketchLib: Enabling efficient sketch-based monitoring on programmable switches,” in *NSDI*, 2022.

[61] L. Yu, J. Sonchack, and V. Liu, “Mantis: Reactive programmable switches,” in *SIGCOMM*, 2020.

[62] M. Bonola, G. Belocchi, A. Tulumello, M. S. Brunella, G. Siracusano, G. Bianchi, and R. Bifulco, “Faster software packet processing on FPGA NICs with eBPF program warping,” in *ATC*, 2022.

[63] Y. Qiu, J. Xing, K.-F. Hsu, Q. Kang, M. Liu, S. Narayana, and A. Chen, “Automated smartnic offloading insights for network functions,” in *SOSP*, 2021.

[64] X. Chen, H. Liu, D. Zhang, Z. Meng, Q. Huang, H. Zhou, C. Wu, X. Liu, and Q. Yang, “Automatic performance-optimal offloading of network functions on programmable switches,” *IEEE Transactions on Cloud Computing*, 2022.

[65] D. Moro, G. Verticale, and A. Capone, “Network function decomposition and offloading on heterogeneous networks with programmable data planes,” *IEEE Open Journal of the Communications Society*, 2021.

[66] F. Pereira, G. Matos, H. Sadok, D. Kim, R. Martins, J. Sherry, F. M. V. Ramos, and L. Pedrosa, “Automatic generation of network function accelerators using component-based synthesis,” in *SOSR*, 2022.

[67] Q. Kang, J. Xing, Y. Qiu, and A. Chen, “Probabilistic profiling of stateful data planes for adversarial testing,” in *ASPLOS*, 2021.

[68] M. D. Wong, A. K. Varma, and A. Sivaraman, “Testing compilers for programmable switches through switch hardware simulation,” in *CoNEXT*, 2020.

[69] J. Yang, Y. Yue, and K. V. Rashmi, “A large scale analysis of hundreds of in-memory cache clusters at twitter,” in *OSDI*, 2020.

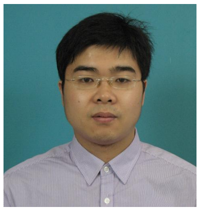
[70] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, “A survey on data plane programming with p4: Fundamentals, advances, and applied research,” *J. Netw. Comput. Appl.*, 2023.

Xueying Zhu is currently a Ph.D. student at Zhejiang University, China. Prior to that, she received her bachelor’s degree from Zhejiang University. Her research interests include in-network computation, SmartNIC, etc.

Yingtao Li is currently an Eng.D. student at Zhejiang University, China. His research interests include in-network computation, programmable switch applications, etc.

Xiang Li Xiang Li is currently a MSE student at Zhejiang University, China. His research interests mainly include in-network computation, RPC optimization, etc.

Jialin Li received his Ph.D. degree from the University of Washington in 2019. Li is currently an Assistant Professor in the School of Computing at the National University of Singapore. His research interests are in co-designing distributed systems with data center networks, data plane operating systems, and system software for programmable network hardware.



Zeke Wang received his Ph.D. degree from Zhejiang University, China in 2011. He is a Research Professor at Collaborative Innovation Center of Artificial Intelligence, Department of Computer Science, Zhejiang University, China. His current research interests mainly focus on building machine learning systems using heterogeneous devices, e.g., SmartNIC and SmartSwitch.